BULLET™

November 23, 1996

Table of Contents

Top Notes

Access-Sharing-Cache Mode

```
Access-Mode (required)
  READONLY         0x00000000   read-only access open
  WRITEONLY        0x00000001   write-only access open
  READWRITE        0x00000002   read/write access open
Share-Mode (required)
  DENYREADWRITE    0x00000010   no other process may share file
  DENYWRITE        0x00000020   no other process may share file for write
  DENYREAD         0x00000030   no other process may share file for read
  DENYNONE         0x00000040   any process may share file
Inherit
  NOINHERIT        0x00000080   child process does not inherit file handles

Cache
  NO_LOCALITY      0x00000000  locality is not known
  SEQ_LOCALITY     0x00010000  access will be mainly sequential
  RND_LOCALITY     0x00020000  access will be mainly random
  MIX_LOCALITY     0x00030000  access will be random with some sequential
  SKIP_CACHE       0x00100000  I/O is not to go through the cache
  WRITE_THROUGH    0x00400000  control returns only after disk is written to
```

Access- and Share-Mode values not explicitly listed are not valid. The file access mode is a combination of ACCESS + SHARE + INHERIT + CACHE. Typical data and index asMode is 0x00000042, though locality may be set accordingly (e.g., 0x00020042 for mostly random access to the file). All values are mapped to the appropriate flags for the OS being used.

The Cache mode options are valid for OPEN_DATA_XB and OPEN_INDEX_XB only; for OPEN_FILE_DOS, the Cache values must be right-shifted by 8. The 'Skip Cache' and 'Write Through' options are not inherited.

Enumerator

The enumerator is a big-endian 16-bit value that serves to differentiate up to 65536 "identical", non-unique keys. It is attached to all keys of DUPS_ALLOWED-flagged index files (set at CREATE_INDEX_XB), and occupies the last two bytes of the key. The first key of the type uses \0\0, the second uses \0\1, and so on. This ordering of bytes is the **reverse** of x86 Intel words, which uses little-endian format.

HIGH-VALUES

HIGH-VALUES signifies a sort value that is the highest possible (sorts last). HIGH-VALUES for a character key would be 0xFF for each byte, or the 256th byte of the collate-sequence table if an NLS sort (which is 0xFF also). For 16-bit signed integer values, 0x7FFF is the highest. And so on...

Tag Field

Bullet uses the term to mean the delete (tag) field, which is the first byte of a DBF record.

*** Notation Under OUT**

References under **OUT** using `*AP.keyPtr` or similar (note the `*`) are used throughout this manual and indicate that Bullet updates the contents at AP.keyPtr with data (i.e., Bullet filled the buffer).

Bullet Functions

The Bullet routines are described in the following sections:

- System Level
- Low-Level Data
- Low-Level Index
- Mid-Level Data
- Mid-Level Index
- High-Level Index+Data
- Network Level
- CP Level

System Level

- INIT_XB
- EXIT_XB
- ATEXIT_XB
- MEMORY_XB
- BACKUP_FILE_XB
- STAT_HANDLE_XB
- GET_ERROR_CLASS_XB
- QUERY_SYSVARS_XB
- SET_SYSVARS_XB
- SET_DVMON_XB
- QUERY_VECTORS_XB
- SET_VECTORS_XB

INIT_XB

Uses INITPACK

IN	OUT
IP.func	IP.stat
IP.JFTsize	IP.versionDOS
	IP.versionBullet
	IP.versionOS
	IP.exitPtr

This must be the first routine called (except for SET_VECTORS_XB). If it has already been called the system variables are restored to their defaults, and an error is returned. Otherwise, the entire Bullet system is initialized, and EXIT_XB is registered with the OS ExitList handler (DosExitList in OS/2).

For more than the default open files (generally 20), set IP.JFTsize to the total number of concurrently open files you need. Depending on your version, Bullet manages up to 1024 Bullet files per process (total data and index; memo files are *not* counted against this total). Setting this less than 20 does nothing. This number is for Bullet files, your files, pipes — anything using a handle. If you need to account for handles that you are managing, you should add those to IP.JFTsize. For example, if you need 10 data files, each with a memo file, and 2 index files per data file, that is 40 total Bullet files. If you need to use 15 other handles, for whatever use, add that number to the 40 Bullet files, for a total setting of 55. The OS also uses 3 handles for itself, so, for all these, IP.JFTsize=58 would be the minimum. You can set it higher, but unused handles are wasted handles. In addition, if the current process has fewer total handles available than the number you specified in IP.JFTsize, Bullet sets the total available handles to IP.JFTsize (as the absolute number of handles required). If the current process already has more total handles than IP.JFTsize, no action is taken.

On return (where no error occurred), the operating system version is in IP.versionDOS (*100) and the Bullet version (*1000) in IP.versionBullet. IP.versionOS return is based on the following table:

Bullet Platform	**IP.versionOS**
MS-DOS	0
Win3x	1
DOSX32	3
OS/2	4
Win32	5

IP.exitPtr returns with the function pointer to the EXIT_XB routine. This function pointer is redundant unless specifically mentioned as being required for your platform. It is not needed in OS/2, but is useful for other platforms, for example, for use with `atexit()`, part of C's standard library.

Win32s, as run in Windows 3.1, returns IP.versionDOS equal to its level. For example, if running Win31, using win32s version 1.25, IP.versionDOS returns as 125, not 310 as might be expected.

EXIT_XB

Uses EXITPACK

IN	**OUT**
`EP.func`	`EP.stat`

Call EXIT_XB before ending your program to release any remaining resources back to the OS. Open files should be closed by using CLOSE_DATA_XB and CLOSE_INDEX_XB. EXIT_XB closes any Bullet files that are still open.

This routine is registered with the operating system and so is called automatically when your program terminates in OS/2, or with the C startup code if `atexit()` is used.

ATEXIT_XB

Uses EXITPACK

IN	**OUT**
`EP.func`	`EP.stat`

This routine is obsolete. In OS/2, the EXIT_XB shutdown procedure is registered with the operating system. *For those system that do not offer this feature in the OS*, the compiler run-time routine `atexit()` is used immediately after calling INIT_XB, using IP.exitPtr as the function pointer for `atexit()`.

MEMORY_XB

Uses MEMORYPACK

IN	OUT
`MP.func`	`MP.stat`
	`MP.memory`

MP.memory returns with the number of bytes in the private memory arena allocatable by the process according to DosQuerySysInfo for QSV_MAXPRMEM, or as provided by the OS. It is for informational use only.

Note: This routine is not mutex-protected.

BACKUP_FILE_XB

Uses COPYPACK

IN	OUT
`CP.func`	`CP.stat`
`CP.handle`	
`CP.filenamePtr`	

Copy an open BULLET index file or data/memo files. BULLET repacks and reindexes files in place, requiring less disk space to perform the function. This routine allows a file to be safely copied for a possible later restore.

This function is *recommended* prior to packing a data file with PACK_RECORDS_XB. For index files, COPY_INDEX_HEADER_XB is sufficient since index files are easily recreated so long as you have the data file along with the index file header.

A full-lock should be in force before copying. A shared lock may be used.

If CP.handle belongs to a DBF data file, and if a memo file is attached, the memo file backup name is as CP.filenamePtr, the backup DBF pathname, but the extension is always set to "._BT". For example, if CP.handle is for a DBF that has a DBT memo file attached, then the current state of the DBF file is copied to CP.filenamePtr, say, "\CURRBACK\ACCT.DBF", and, in this case, the DBT memo file is copied to "\CURRBACK\ACCT._BT". The name of the original DBF/DBT does not matter. If MEMO_EXTENSION of SET_SYSVARS_XB has changed the default, then that extension is used on the memo copy, with '_' replacing its first character.

To *prevent* the backing up of a DBT memo file when backing up a DBF data file, set CP.handle = -CP.handle (i.e., negative CP.handle). This way, the DBT memo file is not copied. To backup only a DBT file, close the DBF and copy the DBT by some other means.

STAT_HANDLE_XB

Uses STATHANDLEPACK

IN	OUT
SHP.func	SHP.stat
SHP.handle	SHP.ID

Get information on a file handle number to determine if it is a BULLET file, and if so, its type: index or data.

SHP.ID	File type	
0	index, IX3	use STAT_INDEX_XB for file stats
1	data, DBF	use STAT_DATA_XB for file stats
-1	unknown	

Only bit0 of SHP.ID is significant if not -1. So, if bit0=0 then the handle belongs to an index file. If bit0=1 then it's a data file.

Memo file handles return as unknown. A DBF file's memo file handle is stored in the DBF file's data area, and is returned by STAT_DATA_XB in SDP.memoHandle.

Note: This routine is not mutex-protected.

GET_ERROR_CLASS_XB

Uses XERRORPACK

IN	OUT
XEP.func	XEP.errClass
XEP.stat	XEP.action
	XEP.location

Get the extended error information for the code passed in XEP.stat. This information includes the error classification, recommended action, and origin of the error.

Any **system** error code can be specified, not necessarily the one that last occurred. If a return code is not a BULLET code, then it is a system error code (from the CP, DosXXX routines).

The ERRCLASS, ERRACT, and ERRLOC items below are OS/2 values, names and descriptions for DosErrClass().

Error Classification

Value	Name	Description
1	ERRCLASS_OUTRES	Out of resources
2	ERRCLASS_TEMPSIT	Temporary situation
3	ERRCLASS_AUTH	Authorization failed
4	ERRCLASS_INTRN	Internal error
5	ERRCLASS_HRDFAIL	Device hardware failure
6	ERRCLASS_SYSFAIL	System failure
7	ERRCLASS_APPEAR	Probably application error
8	ERRCLASS_NOTFND	Item not located
9	ERRCLASS_BADFMT	Bad format for function or data
10	ERRCLASS_LOCKED	Resource or data locked
11	ERRCLASS_MEDIA	Incorrect media, CRC error

12	ERRCLASS_ALREADY	Action already taken or done, or resource exists
13	ERRCLASS_UNK	Unclassified
14	ERRCLASS_CANT	Cannot perform requested action
15	ERRCLASS_TIME	Timeout

Recommended Action

Value	Name	Description
1	ERRACT_RETRY	Retry immediately
2	ERRACT_DLYRET	Delay and retry
3	ERRACT_USER	Bad user input - get new values
4	ERRACT_ABORT	Terminate in an orderly manner
5	ERRACT_PANIC	Terminate immediately
6	ERRACT_IGNORE	Ignore error
7	ERRACT_INTRET	Retry after user intervention

Origin

Value	Name	Description
1	ERRLOC_UNK	Unknown
2	ERRLOC_DISK	Disk
3	ERRLOC_NET	Network
4	ERRLOC_SERDEV	Serial device
5	ERRLOC_MEM	Memory

Note: This routine is not mutex-protected.

QUERY_SYSVARS_XB

Uses QUERYSETPACK

IN	OUT
QSP.func	QSP.stat
QSP.item	QSP.itemValue

Query a BULLET system variable.

To get the function pointers to the sort compare functions, use:

QSP.item	FuncPtr To
ASCII_SORT	ASCII sort compare
NLS_SORT	NLS sort compare
S16_SORT	16-bit signed integer
U16_SORT	16-bit unsigned integer
S32_SORT	32-bit signed integer
U32_SORT	32-bit unsigned integer

All intrinsic sort compares (1-6) point to the same function. They cannot be called except by BULLET itself. The integer compare routines are based on Intel byte order. For Motorola byte order, ASCII sort can be used for all-positive numbers, otherwise a custom sort-compare should be used.

```
10-19        Custom sort-compare functions
```

Before creating or opening an index file with a custom sort-compare function (which is specified during CREATE_INDEX_XB), that function's address must first be sent to BULLET using SET_SYSVARS_XB. Thereafter, that function must be available whenever that index file is accessed. See *Custom Sort-Compare Function* for creating custom sort-compare functions.

To get the function pointers to the build key and expression parser routines, use:

```
QSP.item            FuncPtr To
BUILD_KEY_FUNC      Build key routine
PARSER_FUNC         Key expression parser routine
```

Before creating or opening an index file with a custom build-key or expression parser routine (which is specified at any time, but must be used in a consistent manner), that routine's address must first be sent to BULLET using SET_SYSVARS_XB. Thereafter, that routine should be available since it may be required again. See *Custom Build-Key Routine* for creating a custom build-key routine and *Custom Expression Parser Routine* for creating a custom key expression parser.

To get the BULLET system variables' values, use:

```
QSP.item            Value To
  26   (read-only)  low-word: number of xb$Malloc(), high-word: number of xb$Free()
  27   (read-only)  max instances (2, 32, 999)
  28   (read-only)  max files (100, 250, 1024)
  29  (read-only)   BULLET mutual-exclusion (mutex) semaphore handle

LOCK_TIMEOUT        Lock file region timeout, in milliseconds (default=0)
MUTEX_SEM_TIMEOUT   Mutex semaphore request timeout, in milliseconds (default=0)
PACK_BUFFER_SIZE    Pack buffer size, in bytes (default=0: autosize)
REINDEX_BUFFER_SIZE Reindex buffer size, in bytes (default=0: autosize)
REINDEX_PACK_PCT    Reindex node pack percentage, 50-100% (default=100)
TMP_PATH_PTR        Temporary file path pointer (def=NULL, where TMP= used, then .\)
REINDEX_SKIP_TAG    Reindex tag field character to skip (default=0, no skip)
COMMIT_AT_EACH      Commit each file during INSERT/UPDATE (def=0, defer 'til flush)
MEMO_BLOCKSIZE      Memo file block size (default=512 bytes; minimum is 24 bytes)
MEMO_EXTENSION      Memo file extension (default is 'DBT\0')
MAX_DATAFILE_SIZE   Max data file size-1 (default=2047MB, absolute max is 4095MB)
MAX_INDEXFILE_SIZE  Max index file size-1 (default=2047MB, absolute max is 4095MB)
ATOMIC_MODE         Atomic mode (bit0=1 then atomic Next and Prev access, default=0)
CALLBACK_PTR        Callback routine at reindex/pack (default=0, none)
```

The timeout values determine if the kernel should wait for a pre-determined time before returning an error if the resource cannot be obtained. The lock timeout specifies how long to wait for a lock to be obtained in case some other process has a lock on the same resource. The mutex timeout specifies how long to wait for access to BULLET in case some other *thread* in this process is in BULLET. Multiple processes can access BULLET at the same time, but only one thread in each process can be inside BULLET at any one time.

The buffer sizes, when 0, default to a minimum reasonable size. Performance is acceptable at these sizes. For best performance, provide as much real memory as possible, up to 512KB. Larger buffers can be used.

The reindex node pack percentage determines how many keys are packed on a node. 100% forces as many keys as possible, minus 1.

If the temporary file path pointer is NULL (the default), then the TMP= environment variable is used to locate any temporary files created by BULLET, or if that is not found, then the current directory is used. The pointer supplied, if any, should be to a string containing an existing path (drive should be included; a trailing '\' is optional, but recommended). See REINDEX_XB for size requirements.

The reindex skip tag character, if encountered in the DBF record's tag field (the first byte of each record), causes the reindex routine to not place that record's key value into the index file. Also, BUILD_KEY_XB returns a warning if the record supplied has a matching tag character. To disable skip tag processing, set it to 0.

Inserts and Updates, by default, do not commit each file when that pack is processed. Instead, it is left to the programmer to issue a FLUSH_XB to commit. To force a commit after each pack file is processed, set

CommitAtEach to 1. This is not one single commit, but a commit for each file in the pack, after that file has been processed, but before the next file in the pack is. This will *not* prevent a roll-back should it be needed.

A memo file can have at most 589,822 blocks. At the default 512 bytes per block, that equates to about 288MB. If you need more memo space, increase the block size. The memo extension default is 'DBT\0'. Generally, it's a good idea to leave it at this.

The maximum file sizes are enforced when adding to or reading from DBF files, and when inserting into or reading from index files. The default is 2047MB (0x7FEFFFFF). If your file system permits 4GB files, set the values to 4095MB (0xFFEFFFFF).

The Atomic mode flag determines how key access is handled. When bit0=0, the default, the key routines, NEXT_KEY_XB, PREV_KEY_XB, and GET_NEXT_XB, GET_PREV_XB, use the internal position of the last gotten key as their starting point. In multi-threaded code, it's possible that another thread has since accessed the same file handle and altered the last gotten key. By setting bit0=1, key access (next or previous) can now specify a starting point (typically already set up in AP.keyPtr), rather than starting at the last accessed key for that handle (which may have been changed by another thread).

The Callback routine receives a pointer to a CALLBACKPACK structure on the stack. The callback routine may clean the stack (e.g., a _syscall routine). It may also leave it for the caller to clean (e.g., a _stdcall or __cdecl routine). See bd_rix.c on the distribution disk for an example. When the CALLBACK_PTR == NULL (default), no callback is made. Currently, the callback is made during REINDEX_XB and PACK_RECORDS calls, at a rate dependent on the *_BUFFER_SIZE.

Note: This routine is not mutex-protected.

SET_SYSVARS_XB

Uses QUERYSETPACK

IN	OUT
QSP.func	QSP.stat
QSP.item	QSP.itemValue
QSP.itemValue	

Set a BULLET system variable, returning the previous value.

To use, set QSP.item to the item to set, and QSP.itemValue with the value to use (function's address, variable's timeout value, etc., whatever the case may be). On return, QSP.itemValue is the previous value that QSP.item was set to.

QSP.item	FuncPtr To
ASCII_SORT	ASCII sort compare
NLS_SORT	NLS sort compare
S16_SORT	16-bit signed integer
U16_SORT	16-bit unsigned integer
S32_SORT	32-bit signed integer
U32_SORT	32-bit unsigned integer

All intrinsic sort compares (1-6) point to the same function. They cannot be called except by BULLET itself. They should not be overloaded with custom functions. If you have a custom sort-compare, use one of the custom slots. The integer compare routines are based on Intel byte order. For Motorola byte order, ASCII sort can be used for all-positive numbers, otherwise a custom sort-compare should be used.

```
10-19        Custom sort-compare functions
```

Before creating or opening an index file with a custom sort-compare function (which is specified during CREATE_INDEX_XB), that function's address must first be sent to BULLET using this routine. Thereafter, that function must be available whenever that index file is accessed. See *Custom Sort-Compare Function* for creating custom sort-compare functions.

To set the function pointers to the build key and expression parser routines, use:

QSP.item	FuncPtr To
BUILD_KEY_FUNC	Build key routine
PARSER_FUNC	Key expression parser routine

Before creating or opening an index file with a custom build key or expression parser routine (which is specified at any time, but must be used in a consistent manner), that routine's address must first be sent to BULLET using this routine. Thereafter, that routine should always be ready (in a callable state) since it may be required again. See *Custom Build-Key Routine* for creating a custom build-key routine and *Custom Expression Parser Routine* for creating a custom key expression parser.

To set the BULLET system variables' values, use:

QSP.item	Value To
LOCK_TIMEOUT	Lock file region timeout, in milliseconds (default=0)
MUTEX_SEM_TIMEOUT	Mutex semaphore request timeout, in milliseconds (default=0)
PACK_BUFFER_SIZE	Pack buffer size, in bytes (default=0: autosize)
REINDEX_BUFFER_SIZE	Reindex buffer size, in bytes (default=0: autosize)
REINDEX_PACK_PCT	Reindex node pack percentage, 50-100% (default=100)
TMP_PATH_PTR	Temporary file path pointer (def=NULL, where TMP= used, then .\)
REINDEX_SKIP_TAG	Reindex tag field character to skip (default=0, no skip)
COMMIT_AT_EACH	Commit each file during INSERT/UPDATE (def=0, defer 'til flush)
MEMO_BLOCKSIZE	Memo file block size (default=512 bytes; minimum is 24 bytes)
MEMO_EXTENSION	Memo file extension (default is 'DBT\0')
MAX_DATAFILE_SIZE	Max data file size-1 (default=2047MB, absolute max is 4095MB)
MAX_INDEXFILE_SIZE	Max index file size-1 (default=2047MB, absolute max is 4095MB)
ATOMIC_MODE	Atomic mode (bit0=1 then atomic Next and Prev access, default=0)
CALLBACK_PTR	Callback routine at reindex/pack (default=0, none)

The timeout values determine if the kernel should wait for a pre-determined time before returning an error if the resource cannot be obtained. The lock timeout specifies how long to wait for a lock to be obtained in case some other process has a lock on the same resource. The mutex timeout specifies how long to wait for access to BULLET in case some other *thread* in this process is in BULLET. Multiple processes can access BULLET at the same time, but only one thread in each process can be inside BULLET at any one time.

The buffer sizes, when 0, default to a minimum reasonable size. Performance is acceptable at these sizes. For best performance, provide as much real memory as possible, up to 512KB. Larger buffers can be used.

The reindex node pack percentage determines how many keys are packed on a node. 100% forces as many keys as possible, minus 1.

If the temporary file path pointer is NULL (the default), then the TMP= environment variable is used to locate any temporary files created by BULLET, or if that is not found, then the current directory is used. The pointer supplied, if any, should be to a string containing an existing path (drive should be included; a trailing '\' is optional, but recommended). See REINDEX_XB for size requirements.

The reindex skip tag character, if encountered in the DBF record's tag field (the first byte of each record), causes the reindex routine to not place that record's key value into the index file. Also, BUILD_KEY_XB returns a warning if the record supplied has a matching tag character. To disable skip tag processing, set it to 0.

Inserts and Updates, by default, do not commit each file when that pack is processed. Instead, it is left to the programmer to issue a FLUSH_XB to commit. To force a commit after each pack file is processed, set CommitAtEach to 1. This is not one single commit, but a commit for each file in the pack, after that file has been processed, but before the next file in the pack is. This will *not* prevent a roll-back should it be needed.

A memo file can have at most 589,822 blocks. At the default 512 bytes per block, that equates to about 288MB. If you need more memo space, increase the block size. The memo extension default is 'DBT\0'. Generally, it's a good idea to leave it at this.

The maximum file sizes are enforced when adding to or reading from DBF files, and when inserting into or reading from index files. The default is 2047MB (0x7FEFFFFF). If your file system permits 4GB files, set the values to 4095MB (0xFFEFFFFF).

The Atomic mode flag determines how key access is handled. When bit0=0, the default, the key routines, NEXT_KEY_XB, PREV_KEY_XB, and GET_NEXT_XB, GET_PREV_XB, use the internal position of the last gotten key as their starting point. In multi-threaded code, it's possible that another thread has since accessed the same file handle and altered the last gotten key. By setting bit0=1, key access (next or previous) can now specify a starting point (typically already set up in AP.keyPtr), rather than starting at the last accessed key for that handle (which may have been changed by another thread).

The Callback routine receives a pointer to a CALLBACKPACK structure on the stack. The callback routine may clean the stack (e.g., a _syscall routine). It may also leave it for the caller to clean (e.g., a _stdcall or __cdecl routine). See bd_rix.c on the distribution disk for an example. When the CALLBACK_PTR == NULL (default), no callback is made. Currently, the callback is made during REINDEX_XB and PACK_RECORDS calls, at a rate dependent on the *_BUFFER_SIZE.

Note: Issuing INIT_XB restores all system variables (those setable via this routine) and function pointers, but not vectors, to their default values. This is done even if INIT_XB returns an error that BULLET has already been initialized.

SET_DVMON_XB

This routine is not currently used.

QUERY_VECTORS_XB

Uses QUERYSETPACK

IN	OUT
`QSP.func`	`QSP.stat`
`QSP.item`	`QSP.itemValue`

Query a BULLET OS API vector.

To get the vectors that Bullet makes to access OS API calls, set QSP.item to the desired VECTOR_* constant below. On return, QSP.itemValue has the function pointer for that item.

VECTOR_CLOSE_FILE
VECTOR_CREATE_DIR
VECTOR_CREATE_FILE
VECTOR_CREATE_UNIQUE_FILE
VECTOR_DELETE_FILE
VECTOR_LENGTH_FILE
VECTOR_MOVE_FILE
VECTOR_OPEN_FILE
VECTOR_READ_FILE
VECTOR_SEEK_FILE
VECTOR_UPDATE_DIR_ENTRY
VECTOR_WRITE_FILE
VECTOR_LOCK_FILE
VECTOR_IS_DRIVE_REMOTE
VECTOR_IS_FILE_REMOTE
VECTOR_EXITLIST
VECTOR_REMOVE_EXITLIST
VECTOR_FREE
VECTOR_GET_SORT_TABLE
VECTOR_GET_COUNTRY_INFO
VECTOR_GET_ERROR_CLASS
VECTOR_GET_MEMORY
VECTOR_GET_TMP_DIR
VECTOR_GET_VERSION
VECTOR_MALLOC
VECTOR_SET_HANDLE_COUNT
VECTOR_GET_TIME_INFO
VECTOR_UPPERCASE
VECTOR_CLOSE_MUTEX_SEM
VECTOR_CREATE_MUTEX_SEM
VECTOR_RELEASE_MUTEX_SEM
VECTOR_REQUEST_MUTEX_SEM

If QSP.itemValue is returned NULL, the default Bullet OS API call is being used.

Note: This routine is not mutex-protected.

SET_VECTORS_XB

Uses QUERYSETPACK

IN	OUT
`QSP.func`	`QSP.stat`
`QSP.item`	`QSP.itemValue`
`QSP.itemValue`	

Set a BULLET OS API vector.

To set the vectors that Bullet makes to access OS API calls, set QSP.item to the desired VECTOR_* constant (see QUERY_VECTORS_XB for the list), and QSP.itemValue to its replacement's address. On return, QSP.itemValue has the previous function pointer for that item.

Example replacement routines are in `ccdosfn.c`. Many of the routines in `ccdosfn.c` are standard C, but some are OS-specific and must be updated for the OS being used.

If QSP.itemValue is set to NULL, the default Bullet OS API call is used.

On return from a successful call, QSP.itemValue is the value of the previous function pointer for that item, which may be returned NULL, indicating that the default Bullet OS API call was being used.

Low-Level Data

- `CREATE_DATA_XB`
- `OPEN_DATA_XB`
- `CLOSE_DATA_XB`
- `STAT_DATA_XB`
- `READ_DATA_HEADER_XB`
- `FLUSH_DATA_HEADER_XB`
- `COPY_DATA_HEADER_XB`
- `ZAP_DATA_HEADER_XB`

CREATE_DATA_XB

Uses CREATEDATAPACK

```
    IN                       OUT
CDP.func                 CDP.stat
CDP.filenamePtr
CDP.noFields
CDP.fieldListPtr
CDP.fileID
```

Create a new BULLET DBF data file with the name at CDP.filenamePtr, and an optional DBT memo file.

Before using this routine, allocate an array of field descriptors of type FIELDDESCTPYE, one for each field in the record (number of fields as set in CDP.noFields). It is *recommended* that this allocation be zeroed before use since fieldnames and reserved entries must be 0-filled:

```
FIELDDESCTYPE fieldList[12];  // 12 fields used in data record
     :
   memset(fieldList,0,sizeof(fieldList)); // init unused bytes to 0 (required)
```

Filename

The drive and path must exist if used as part of the filename. Long filenames may be used if supported by the file system in use. As with all text strings used by Bullet, the filename must end in a '\0'.

Number of Fields

The number of descriptors in the array, described next. Each field has a descriptor. The tag field is not a formal field, and so has no descriptor, and is not counted in the number of fields. The maximum number of fields is 254 according to the DBF standard. Bullet allows 1024, but 254 should be used if creating a standard DBF III Plus file.

Field Descriptors

For each field, a descriptor is used to identify and type it. These descriptors are assigned to an array; the pointer to that array is assigned to CDP.fieldListPtr. The format of the descriptor follows, with a physical format listed in *DBF File Format*.

Fieldname

10 characters plus null byte terminator. Valid fieldname characters are ASCII A-Z (upper-case) and the underscore (ASCII 95). All bytes after the fieldname must be null bytes. E.g., if the fieldname is "LNAME", five characters, the following six bytes (including the 11th byte) are set to 0. The eleventh byte is always a null byte since 10 characters is the maximum fieldname length. Extended ASCII characters (above 127) should not be used.

```
fieldList[0].fieldname = "ANYNAME";   // see memset() above
```

Field type and size

Standard Xbase field types are C, D, L, M, and N:

Type **Description**

C Character field, any code page character, 1 to 255 characters.

Null bytes are not desirable except as a string terminator. There is no requirement that field data be terminated with a '\0'. The field data should be left-justified within the field, but this is not required (in which case use leading spaces, not 0 bytes).

```
fieldList[0].fieldType = 'C';
fieldList[0].fieldLen = 25;
fieldList[0].fieldDC = 0;
```

D Date field, valid ASCII digits for date, 8 characters.

The physical format is YYYYMMDD, where YYYY is the year (1999), MM is the month (1-12), and DD the day (1-31). The date field is always 8 bytes long, and is in ASCII digits '19991231'). If no date, set to all spaces.

```
fieldList[0].fieldType = 'D';
fieldList[0].fieldLen = 8;
fieldList[0].fieldDC = 0;
```

L Logical field, <SPACE> Y N T F y n t f, 1 character.

A single-byte field. When not yet initialized the value will be a <SPACE> (ASCII 32). This is typically displayed as a '?' to the user, indicating that the field has not been initialized. Initialized values are variations of yes, no, true, false ('Y', 'y', etc.).

```
fieldList[0].fieldType = 'L';
fieldList[0].fieldLen = 1;
fieldList[0].fieldDC = 0;
```

M Memo field, 10 ASCII digits, 10 characters.

Field data is used as the block number in the corresponding DBT memo file. Each block is typically 512 bytes, with the first block (block #0) used as the memo file header. If no block is used in the .DBT by this record, the field is set to <SPACES>. The first memo block is stored as '0000000001'. (This description is valid for dBASE IV and later memo files, as created and used by BULLET.) Some Xbase versions use field types B and G as variations of memo files. They are as M, but contain general data (as in anything), while memo files contain only text. BULLET supports any type data in its memo files, and you may use the CDP.fieldType of 'B' or 'G'.

More than one memo field per record is permitted. For example, you may need a memo for the printable address, where the address is free-form rather than in separate fields (i.e., you have both forms), and another memo for general notes, and yet a third for problem reports, and so on. All these, and all memos for the rest of the DBF file, are stored in the same DBT memo file.

Note: BULLET does not use the fieldType with regard to identifying memo field type; it is the programmer's responsibility to check the fieldType and act on it accordingly, such as adding memo numbers.

```
fieldList[0].fieldType = 'M';
fieldList[0].fieldLen = 10;
fieldList[0].fieldDC = 0;
```

N Numeric field, ASCII digits, 19 digits maximum (see below).

All standard Xbase data is stored in ASCII form (for universal exchange). Numeric fields are to be right-justified, with leading spaces, and an aligned decimal point, if any (relative this field in other records). Do not end the field with a null byte.

The total size of the numeric field is specified in .fieldLen, which includes any leading sign, the decimal point, and decimal digits to the right of the decimal point (if any decimal point). The maximum total size is 19 places. If a decimal point, then the number of digits to the right may be from 1 to 15 digits, but must be no more than the total-2.

```
FieldLen.FieldDC    Example
       8.2         ' 2345.78'
       8.2         '12345.78'
       8.2         '-2345.78'
       8.1         '123456.8'
       8.0         '12345678'
       5.3         '2.235'
       5.4         (not valid)

fieldList[0].fieldType = 'N';
fieldList[0].fieldLen = 8;
fieldList[0].fieldDC = 2;
```

Although not dBASE compatible, you may use **binary fields** in your data records. The Xbase standard always has ASCII data in the data fields, even if the field is numeric. For example, an 'N' type field of 8.2 (total length.decimal-count) 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 'Y' (actually, anything but an 'N') and, *if it is to be used as a key field*, also set the sort function to the appropriate type (S16_SORT, etc.). The field length (fieldList[x].fieldLen) for a 'Y' field type is 2 if 16-bit, and 4 if 32-bit. Also possible is floating-point (with a custom sort-compare function). A likely field type marker for this would be 'F'. Note that both 'Y' and 'F' are completely non-standard Xbase types, and only your programs will understand them.

Note: 'B' should not be used as a binary field type marker since dBASE V uses 'B' to signify a binary-data memo file field. Bullet makes no distinction in its memo file data; anything can be placed in them. Typically, your memo fields are marked as 'M' in Bullet, but could also be 'B' or 'G'.

File ID

Conventional dBASE DBF files have a CDP.fileID=3. To create a memo file (DBT, dBASE IV compatible), set CDP.fileID=x8B. For the DBT to be created, both bits 3 and 7 (0x88) must be set. The other bits may be anything, and are not checked.

In creating your DBF files, specify CDP.fileID=3 to ensure compatibility across Xbase versions, and limit record length to 4000 characters. If creating a non-standard DBF (e.g., non-standard field types, extended field lengths, etc.) it's recommended to use CDP.fileID=0 or CDP.fileID=1. For a standard DBF file with a memo file (dBASE IV or later), use CDP.fileID=0x8B (eight-bee).

Generally, field data is space-filled. String terminators are allowed in C-haracter field types, but should not be used in other fields.

Memo File Creation

If bits 3 and 7 are set in CDP.fileID, a memo file is created for the DBF. The memo filename will be the same as the DBF name except the extension. The memo file is created after the DBF, with a block size of 512 bytes, and filename extension of '.DBT'. The default block size and extension can be overridden (see SET_SYSVARS_XB) prior to calling this routine.

Note: A simple way to check that your record description in the field list matches your source code structure is to compare the number of bytes used by all fields in the field list (+1 for the delete tag byte) with the size of your program's structure. They must equal. Also verify each field's size to make sure they match. By doing this you prevent the problem where data on disk does not match the data in your record structure.

OPEN_DATA_XB

Uses OPENPACK

IN	OUT
`OP.func`	`OP.stat`
`OP.filenamePtr`	`OP.handle`
`OP.asMode`	

Open an existing DBF data file for use. For DBF opens, two parameters are specified: the filename and the access-sharing mode. The OP.xbLink parameter is used only for index opens, and so is not used here.

The OP.asMode has optional cache mode settings. The caching modes cover locality, write-through, and skip cache. Locality is typically mostly random (RND_LOCALITY), but may be mostly sequential if the data file has been sorted and the index file recently reindexed and processing is mostly in-order (first to last, rather than random). Locality is used to tune the cache. Also, normally, data is written to the cache with control returning immediately to the program before the disk is written (an asynchronous write). To force the write to take place before control is returned (a synchronous write), use the WRITE_THROUGH mode. To skip the cache completely, use the SKIP_CACHE mode. This, as all OP.asMode settings, affects this file handle only. asMode flags are in the Top Notes section of this manual.

On a successful open, the file handle is returned. Use this handle for all further access to this file. If the DBF was created with a compatible **memo file**, it is also opened. The handle of the memo file is available via STAT_DATA_XB, but all access to the memo file is made with the handle of the memo file's master DBF (the handle returned by this routine in OP.handle). The memo file is opened using the same OP.asMode.

Note: FoxPro DBF files with Fox memo files (FPT) use an ID of 0xFF. Bullet does not support FoxPro memo files, and so opening a FoxPro DBF with a Fox memo file returns the warning message, WRN_CANNOT_OPEN_MEMO. The DBF file is opened, and the warning can be ignored.

Once open, you can get information on the data file by using STAT_DATA_XB.

Each DBF data file opened allocates and commits at least 4K bytes for internal use (this will vary if SET_VECTORS_XB is used for VECTOR_MALLOC):

Number of Fields	Memory
1 to 121	4KB
122 to 249	8KB
each 128 fields	4KB more

This memory is released when you close the file with CLOSE_DATA_XB. or issue EXIT_XB.

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

When BULLET creates a DBF, it forces all fieldnames to upper-case (it's a DBF requirement) and 0-fills them as well. On data file opens (OPEN_DATA_XB), it also does this, and so any header copy (COPY_DATA_HEADER_XB) will have upper-cased fieldnames (the original file is not changed). To prevent BULLET from mapping the fieldnames to upper-case (NLS mapping, though fieldnames should be standard ASCII characters only), set bit31 of OP.asMode to 1 (0x80000042, for example). This skips the case mapping. Zero-filling always takes place, and starts after the first '\0' byte in the fieldname.

CLOSE_DATA_XB

Uses HANDLEPACK

IN	OUT
HP.func	HP.stat
HP.handle	

Close an existing data file.

Closing the file updates the file header and releases the memory used by the file. Any associated memo file is closed, too. Any outstanding locks should be unlocked before calling this routine.

Note: Remaining locks belonging to this handle are released by the OS upon the successful close.

STAT_DATA_XB

STATDATAPACK

IN	OUT	
SDP.func	SDP.stat	SDP.recordLength
SDP.handle	SDP.fileType	SDP.xactionFlag
	SDP.flags	SDP.encryptFlag
	SDP.progress	SDP.herePtr
	SDP.morePtr	SDP.memoHandle
	SDP.fields	SDP.memoBlockSize
	SDP.asMode	SDP.memoFlags
	SDP.filenamePtr	SDP.memoLastRecord
	SDP.fileID	SDP.memoLastSize
	SDP.lastUpdate	SDP.lockCount
	SDP.records	

Return information BULLET has on the DBF data file specified by SDP.handle.

Item	**Description**
stat	Return code of operation
fileType	1 for DBF
flags	Bit0=1 if file has changed since last flush (dirty)
	Bit1=1 if the file has its entire region locked (full lock)
	Bit2=1 if the file has a shared lock in use (cannot write to it if so)
progress	Percentage of pack operation completed, 1-99, or 0 if done
morePtr	Always 0
fields	Number of fields per record (does not included implicit tag field)
asMode	Access-sharing-cache mode as specified at open (excludes NoCaseMap bit31)
filenamePtr	Pointer to the filename as used in OPEN_DATA_XB
fileID	ID byte used when the DBF was created (the first byte of the file)
lastUpdate	Date of last change (binary: high word=year (1999), low byte=day, high byte=month)
records	Number of records in the DBF (includes any delete-tagged records)
recordLength	Total length of a data record, including tag field
xactionFlag	Not currently used
encryptFlag	Not currently used
herePtr	Pointer to the internal data control area for this file handle
memoHandle	Handle of open memo file (0 if none)
memoBlockSizeMemo	file block size (512 is typical, 24 is minimum)
memoFlags	Bit0=1 dirty
memoLastRecord	Last accessed memo record (0 if none; same as 'block number')
memoLastSize	Size of last accessed memo record (in bytes, including 8 bytes overhead)
lockCount	Number of full-locks in force (locked on first, unlocked on last)

Typically, your program tracks whether a particular handle belongs to an index file or data file. In cases where this is not possible, call the STAT_HANDLE_XB routine to determine what file type a handle is.

Note: In network environments, you should have an exclusive lock on the data file (and implicitly, therefore, the memo file, if any) before using this routine to ensure that the information is current. This also applies to multi-process environments on a single machine. This routine is not mutex-protected. During the call, the file handle must not be closed by another thread.

READ_DATA_HEADER_XB

Uses HANDLEPACK

IN	OUT
`HP.func`	`HP.stat`
`HP.handle`	

Reload the disk copy of the data header for the opened DBF data file handle, refreshing the in-memory copy. Any associated memo file is refreshed, too.

Normally, this routine is not called directly but rather is done automatically when you full-lock the file (LOCK_XB). This routine does not refresh the header if the current state is dirty (SDP.flags, bit0=1); it returns an error if tried.

Since it is recommended that a full-lock be in force *before* using this routine (shared or exclusive), and since a full-lock always reloads the header anyway, calling this routine should never be required. If ever there is a reason to use this routine without having a full-lock in force, then, of course, you may need to. However, it is not wise to reload the header without a full-lock (which locks the header). If you are using your own lock routines, this call will be very useful.

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 this. When BULLET full-locks a file, BULLET automatically reloads the header by using this routine. When removing the full-lock, BULLET automatically flushes the header using FLUSH_DATA_HEADER_XB (unless the current lock is a shared lock (SDP.flags bit2=1)).

FLUSH_DATA_HEADER_XB

Uses HANDLEPACK

IN	OUT
`HP.func`	`HP.stat`
`HP.handle`	

Write the in-memory 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 (the dirty bit is set). Any associated memo file is flushed, too. This routine ensures that the data header on disk matches exactly the data header that is being maintained by BULLET.

Normally, this routine is not called directly but rather is done automatically when you unlock the file (UNLOCK_XB). This routine does not write out the header if the current lock state is shared (SDP.flags, bit2=1); it returns an error if tried. Unlocking a full-lock performs a flush automatically, and so you may never need to explicitly call this routine. Also, when relocking from an exclusive full-lock to a shared full-lock, an automatic flush is performed.

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 image of the header would not get updated since the disk image is written only on a CLOSE_ or FLUSH_DATA_HEADER_XB, or on EXIT_XB (and also prior to PACK_RECORDS_XB). After the system restarts, BULLET opens the file, reads the header and thinks that there are 100 records. You lost a record. Now, if after that record add your program issues FLUSH_DATA_HEADER_XB, the header on disk is refreshed with the in-memory copy, keeping the two in sync. This routine also updates the directory entry for the file, keeping things neat there (file size). Still, it doesn't come without cost: flushing takes additional time, therefore, you may elect to flush periodically, or whenever the system is idle.

Note: You should have a full-lock on the file before using this routine.

COPY_DATA_HEADER_XB

Uses COPYPACK

IN	OUT
`CP.func`	`CP.stat`
`CP.handle`	
`CP.filenamePtr`	

Copy the DBF file structure of an open data file to a new 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 CREATE_DATA_XB. The resultant DBF will be exactly like the source, including number of fields and field descriptions, and an empty memo file, if applicable. It contains 0 records. It may be opened as a regular Bullet data file.

A typical use for this is to create a work file, where only a subset of records is required. For example: You want to process all records of those whose last name starts with A. Copy the header to a work file, use GET_XB routines to get records meeting the criterion, writing those that fit the criterion to the work file (using either Add/Reindex, or Insert). A new index can be specified, or an existing index can be copied using COPY_INDEX_HEADER_XB.

ZAP_DATA_HEADER_XB

Uses HANDLEPACK

IN	OUT
`HP.func`	`HP.stat`
`HP.handle`	

Delete all records in a DBF data file.

This routine is similar to COPY_DATA_HEADER_XB except for one major difference: **All data records in the source file are physically deleted**. No action is performed on the DBF's memo file, if any.

If you have a DBF file with 100 records and use ZAP_DATA_HEADER_XB on it, all 100 records will be physically deleted and the file truncated as if no records were ever in the file. All data records are lost forever.

Low-Level Index

- CREATE_INDEX_XB
- OPEN_INDEX_XB
- CLOSE_INDEX_XB
- STAT_INDEX_XB
- READ_INDEX_HEADER_XB
- FLUSH_INDEX_HEADER_XB
- COPY_INDEX_HEADER_XB
- ZAP_INDEX_HEADER_XB

CREATE_INDEX_XB

Uses CREATEINDEXPACK

IN	OUT
CIP.func	CIP.stat
CIP.filenamePtr	
CIP.keyExpPtr	
CIP.xbLink	
CIP.sortFunction	
CIP.codePage	
CIP.countryCode	
CIP.collatePtr	
CIP.nodeSize	

Create a new BULLET index file.

Before you can create an index file, you must first have opened (and have created if necessary) the BULLET DBF data file that it is to index. To open the data file, use OPEN_DATA_XB. To create the index file, you need to provide the name to use, the key expression, the DBF file link handle (obtained from the OPEN_DATA_XB call), sort function/flags, and optionally, the code page, country code, and collate table. There's also a node size parameter. Select 512, 1024, or 2048 bytes.

Note: BULLET has an optional *external* data mode where only indexing is done — no data file link is used. In this mode, BULLET manages the index files of the key and key data you provide (key data is any 32-bit item, e.g., a record number, offset, etc.). This would be useful for indexing non-DBF files, even files with variable-length records.

Filename

The drive and path must exist if used as part of the filename. Long filenames may be used if supported by the file system in use.

Key Expression

The key expression is an ASCIIZ 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. Up to 16 component parts can be used in the expression.

Two functions are supported in evaluating a key expression. These are SUBSTR() and UPPER():

SUBSTR() extracts part of a field's data 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.

Any name used in the key expression must be a valid field name in the DBF data file. Below are a few sample key expressions for the given data file structure:

Name	Type	Len	DC
FNAME	C	25	0
LNAME	C	25	0
SSN	C	9	0
DEPT	N	5	0

A few example key expression strings for this structure:

```
keyExpression[]="LNAME";
keyExpression[]="LNAME+FNAME";
keyExpression[]="SUBSTR(LNAME,1,4)+SUBSTR(FNAME,1,1)+SUBSTR(SSN,6,4)";
keyExpression[]="UPPER(LNAME+FNAME)";  // for ASCII sort function only
keyExpression[]="DEPT+SSN";
```

In the last example above, even though DEPT is a numeric field type (N), it can still be used as a component of a multi-part character key with SSN (whose type is set to character). This because numeric fields in dBASE DBF data files are ASCII digits, not binary values, and are sorted according to the ASCII value or NLS weight.

The key expression is parsed when the index file is created (this routine) and also when reindexed (REINDEX_XB). The parser() function, which parses the key expression, may be replaced by a programmer-supplied function if additional functionality is needed. See *Custom Expression Parser Routine* for details.

DBF File Link Handle (xbLink)

Since BULLET evaluates the key expression when the file is created (this routine) or during reindex, it must have access to the DBF file to verify that the key expression is valid. You must, therefore, supply the OS file handle of the opened DBF data file. If you later change the structure of the DBF data file (add new fields, remove others, etc.), you must use the reindex routine to re-evaluate the key expression. If the key expression is no longer valid after the data file changes (key field has changed names, etc.), then you must create a brand new index file with this routine, supplying the new key expression, rather than reindexing.

Note: Handles 0-2 are reserved handles and should never be used for any BULLET routine. Also, .xbLink of -1 is reserved by BULLET to indicate an *external data* index for index create and open routines.

Sort Function

The sort function specifies the sort method for the index file. Essentially, this defines the compare function used by the access methods employed by BULLET when doing any type of key access (reading and writing). There are six intrinsic sort compare functions available, with an additional 10 sort compare functions that can be specified by the programmer (see *Custom Sort-Compare Function*).

While not recommended, duplicate key values are supported and managed by BULLET. The flag DUPS_ALLOWED is OR'ed with the sort function value to specify this. Generally, it is not acceptable to allow duplicate keys for an index; there should be one key identifying one record without any further investigation needed to determine if the key is indeed for that record. This is not possible, not consistently so, when duplicate keys exist. It is much simpler to define your key so that duplicates are not generated, than it is to deal with duplicate keys once you have them. If an attempt to insert a key that already exists in the index file is made, and DUPS_ALLOWED was not specified when the index file was created, the insert fails (either a STORE_KEY_XB, an INSERT_XB, or a REINDEX_XB operation), and error EXB_KEY_EXISTS is returned.

For Windows, the sort function flag, USE_ANSI_CHARSET may be specified. This instructs Bullet to use the ANSI character set (i.e., Windows character set) for the current system code page. The default is USE_OEM_CHARSET, which is used by MS-DOS and OS/2.

Only data contained within a record should be used to build a key. The physical record number is not part of the data of a record since it can change at any time without you knowing about it (during a pack, for example). Do not use the record number in an attempt to generate unique keys. Only use what is available in the data record itself, so that the key can be built, or rebuilt, at any time.

The intrinsic sort compare functions of BULLET are:

ASCII_SORT	1 - ASCII (up to 16 key components)
NLS_SORT	2 - NLS (up to 16 key components)
S16_SORT	3 - 16-bit signed integer (single component)
U16_SORT	4 - 16-bit unsigned integer (single component)
S32_SORT	5 - 32-bit signed integer (single component)
U32_SORT	6 - 32-bit unsigned integer (single component)

To expand on the basic functionality provided by BULLET, you can supply your own parser, build, and sort compare routines, and have BULLET use them instead. With your own routines in place, you can have BULLET do just about anything with regard to the index file, including evaluating the key expression dynamically; using more components; allowing multi-part binary keys; and more.

Generally, character data (type C) is left-justified, and unused space is padded with the SPACE character (ASCII 32). It is permissible to use C-type strings, or to 0-fill unused space.

Numeric data (type N) is right-justified, with leading space to be padded with the SPACE character. It is **not** permissible to use 0-fill leading bytes (literal '0' can used, however). Since the field is right-justified, it is not generally desirable to terminate the field with a 0 byte, either. If a decimal count is specified (not 0), the decimal point location is to be the same for all entries in this field. The field description must match the actual data: If the field length and field decimal count was specified as 10.2 (10 total bytes, 2 digits to the right of the decimal, then the data is to be formatted so that '-234567.90' is the longest data that is to be entered in that field. All entries in all records for this field must be of the same format. For example, ' 987654.21', or '   23.01', or '   -1.99' (note the leading spaces). Numeric data is indexed as ASCII values (i.e., the key remains character digits) unless a binary sort function is specified.

Using one of the binary integer sort compare functions requires the following:

1. Single component expression.
2. Field type must be N if the field has ASCII digits, or **if the data is binary**, then the field type must be Y (actually, anything but N).
3. If ASCII digits, the value must fit into the function size (-32768 to 32767 or 0-65535 for signed/unsigned 16-bit 2,147,483,647 to -2,147,483,648 or 0-4,294,967,295 for 32-bit signed/unsigned values).

Although not dBASE compatible, you may use **binary fields** in your data records. The Xbase standard always has ASCII data in the data fields, even if the field is numeric. For example, an 'N' type field of 8.2 (total length.decimal-count) is stored as an ASCII text string in the data record, say, a string like ' 1100.55' (there is no \0 string terminator). If you want dBASE compatibility, your field data must 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 'Y' (actually, anything but an 'N') and, *if it is to be used as a key field*, also set the sort function to the appropriate type (S16_SORT, etc.). The field length (fieldList[x].fieldLen) for a 'Y' field type is 2 if 16-bit, and 4 if 32-bit. For 64-bit integers, a custom sort-compare function is required since there is no intrinsic 64-bit function available.

Note: 'B' should not be used as a binary field type marker since dBASE V uses 'B' to signify a binary-data memo file field. Bullet makes no distinction in its memo file data; anything can be place in them. Typically, your memo fields are marked as 'M' in Bullet, but could also be 'B' or 'G'.

The key expression string you specify may be up to 159 characters, and evaluate out to 64 bytes (62 bytes if DUPS_ALLOWED is specified). The expression string must be 0-terminated, as are all strings used by BULLET itself (filenames, etc.).

National Language Support (NLS)

National Language Support is available to properly sort most languages' alphabets. BULLET uses NLS to build the collate sequence table (sort table) that it uses to ensure proper sorting of mixed-case keys as well as the sorting of foreign language alphabets which use extended-ASCII. In order for BULLET to use the proper collate table, it must know what code page and country code to use. If not supplied, Bullet gets this information directly from the OS, which provides the cc/cp for the current process. If you supply cc/cp, the code page must be loaded or an error is returned (see OS/2's CHCP command). The collate table generated is made part of the index file so that all subsequent access to the index file maintains the original sort order, even if, say, the MIS shop is moved to another location/computer system using another country code/code page. These three items are discussed below.

Code Page

To use the default code page of the current process, specify a code page of 0. The OS is queried for the current code page and this code page is then used. Any valid and available code page can be specified. This is used only if a custom sort-compare or NLS sort is specified.

Country Code

To use the default country code of the current process, specify a country code of 0. The OS is queried for the current country code and this code is then used. Any valid country code can be specified. This is used only if a custom sort-compare or NLS sort is specified.

Custom Collate Table

If a null-pointer is specified, and a *custom sort-compare* or *NLS sort* is specified, BULLET queries the OS for the collate sequence table to use based on the code page and country code specified. Otherwise, the supplied table is used. Intrinsic sorts *other* than NLS use no collate table, and the country code or code page are not used, either.

To use a sort weight table of your own choosing, supply a non-NULL pointer to this parameter. If non-NULL, the passed table is used for sort compares. The table is composed of 256 weight values, one per character. For example, table position 65 ('A') and table position 97 ('a') could both be weighted 65, so that each are considered equal when sorted. If a custom sort-compare function was specified, this sort table may, or may not, be used — it depends on whether the sort compare function uses the table (it's all up to the custom sort-compare function's logic).

Typically, you set both the code page and country code = 0, and the collate table pointer to NULL. See *Sort Function*, above, for using USE_ANSI_CHARSET as a sort function flag, especially for Windows.

Node Size

The index file is read and written in node-size chunks. The larger the node size, the more keys are read or written per chunk. Generally, a smaller node size offers better random key access, while a larger node size offers better sequential key access.

Typically, an average node utilizes 66% of the node space for keys (a very small number may contain only a few keys, while some may be filled completely). In a 512-byte node file, for a key length of 8, there is room for (512-5)/(keylength+8) nodes, or 31 keys. Since a typical node is filled to 66%, that means about 20 keys per node. For a 2048-byte node file, same parameters, there is room for (2048-5)/(keylength+8), or 127 keys. At the standard 66% load, there are typically 83 keys per 2K node. That's 3 more keys per 2K of disk than the 512-byte node gives for 4 nodes (20 keys*4), The trade-off is that each node is 4 times as large, and so requires 4 times more searching. Actual performance differences may be minimal, or may be great. Run tests on expected data to determine the best for the data and access used.

OPEN_INDEX_XB

Uses OPENPACK

IN	OUT
`OP.func`	`OP.stat`
`OP.filenamePtr`	`OP.handle`
`OP.asMode`	
`OP.xbLink`	

Open an existing index file for use. For index opens, three parameters are specified: the filename, the access-sharing mode, and the handle of the open DBF file that this file indexes. It is required to open the data file before you can open its related index file.

Note: Handles 0-2 are reserved handles and should never be used for any BULLET routine. Also, .xbLink of -1 is reserved by BULLET to indicate an *external data* index for index create and open routines.

On a successful open, the file handle is returned. Use this handle for all further access to this file.

Once open, you can get information on the index file by using STAT_INDEX_XB.

Each index file that you open allocates and commits 4K bytes for internal use (this will vary if SET_VECTORS_XB with VECTOR_MALLOC is used). This memory is released when you close the file with CLOSE_INDEX_XB or issue EXIT_XB, or your program terminates.

The OP.asMode has optional cache mode settings. The caching modes cover locality, write-through, and skip cache. Locality is typically mostly random (RND_LOCALITY), but may be mostly sequential if the data file has been sorted and the index file recently reindexed and processing is mostly in-order (first to last, rather than random). Locality is used to tune the cache. Also, normally, data is written to the cache with control returning immediately to the program before the disk is written (an asynchronous write). To force the write to take place before control is returned (a synchronous write), use the WRITE_THROUGH mode. To skip the cache completely, use the SKIP_CACHE mode. This, as all OP.asMode settings, affects this file handle only.

BULLET has an optional *external data* mode where only indexing is done — no data file link is used. In this mode, BULLET manages the index files of the key and key data you provide (key data is any 32-bit item, e.g., a record number, offset, etc.). This would be useful for indexing non-DBF files, even files with variable-length records. Only those routines that do not access the data file may be used (any routine using AP.recPtr, for example, could not be used, but NEXT_KEY_XB may).

CLOSE_INDEX_XB

Uses HANDLEPACK

IN	OUT
`HP.func`	`HP.stat`
`HP.handle`	

Close an open index file.

Closing the file updates the file header and releases the memory used by the file. Any outstanding locks should be unlocked before calling this routine.

Note: Remaining locks belonging to this handle are released by the OS upon the successful close.

STAT_INDEX_XB

Uses STATINDEXPACK

IN	OUT	
`SIP.func`	`SIP.stat`	`SIP.keyLength`
`SIP.handle`	`SIP.fileType`	`SIP.keyRecNo`
	`SIP.flags`	`SIP.keyPtr`
	`SIP.progress`	`SIP.herePtr`
	`SIP.morePtr`	`SIP.codePage`
	`SIP.xbLink`	`SIP.countryCode`
	`SIP.asMode`	`SIP.CTptr`
	`SIP.filenamePtr`	`SIP.nodeSize`
	`SIP.fileID`	`SIP.sortFunction`
	`SIP.keyExpPtr`	`SIP.lockCount`
	`SIP.keys`	

Return information BULLET has on the index file specified by SIP.handle.

Item	Description
stat	Return code of operation
fileType	0 for index, IX3
flags	Bit0=1 if file has changed since last flush (dirty) Bit1=1 if the file has its entire region locked (full lock) Bit2=1 if the file has a shared full-lock in use (cannot write to it if so)
progress	Percentage of reindex operation completed, 1-99, or 0 if done
morePtr	Always 0
xbLink	Handle of the open DBF file this file indexes
asMode	Access-sharing-cache mode as specified at open
filenamePtr	Pointer to the filename as used in OPEN_INDEX_XB
fileID	'31ch' (the first four bytes of the file)
keyExpPtr	Pointer to the key expression used when the index was created
keys	Number of keys in the index file
keyLength	Length of the key, including 2-byte enumerator if DUPS_ALLOWED
keyRecNo	The DBF record number that the last accessed key indexes
keyPtr	Pointer to the last accessed key (valid for keyLength bytes)
herePtr	Pointer to the internal data control for this file
codePage	Code page used when the index file was created (the actual code page)
countryCode	Country code used when the index file was created (the actual country code)
CTptr	Pointer to the collate sequence table used for NLS sorting (each index file has its own sequence table) or NULL if not an NLS index
nodeSize	Size of an index node, 512, 1024, or 2048 bytes, as specified during create
sortFunction	The index sort-compare method (low word) and the sort flags (high word), with sort-compare values 1-9 being intrinsic, and 10-19 being custom functions; the DUPS_ALLOWED flag is bit0 of the high word (allowed if set)
lockCount	Number of full-locks in force (locked on first, unlocked on last)

Typically, your program tracks whether a particular handle belongs to an index file or a data file. In cases where this is not possible, call the STAT_HANDLE_XB routine to determine what file type a handle is.

Note: In network environments, you should have an exclusive lock on the index file before using this routine to ensure that the information is current. This routine is not mutex-protected. During the call, the file handle must not be closed by another thread.

READ_INDEX_HEADER_XB

Uses HANDLEPACK

IN	OUT
`HP.func`	`HP.stat`
`HP.handle`	

Reload the disk copy of the index header for the opened index file handle, refreshing the in-memory copy.

Normally, this routine is not called directly but rather is done automatically when you full-lock the file (LOCK_XB). This routine does not refresh the header if the current state is dirty (SIP.flags, bit0=1); it returns an error if tried.

Since it is recommended that a full-lock be in force *before* using this routine (shared or exclusive), and since a full-lock always reloads the header anyway, calling this routine should never be required. If ever there is a reason to use this routine without having a full-lock in force, then, of course, you may need to. However, it is not wise to reload the header without a full-lock (which locks the header). If you are using your own lock routines, this call will be very useful.

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: An index file has 100 keys. Two programs access this index 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 index 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 this. When BULLET full-locks a file, it automatically reloads the header using this routine. When removing the full-lock, BULLET automatically flushes the header using FLUSH_INDEX_HEADER_XB (unless the current lock is a shared lock (SIP.flags bit2=1)).

FLUSH_INDEX_HEADER_XB

Uses HANDLEPACK

IN	OUT
`HP.func`	`HP.stat`
`HP.handle`	

Write the in-memory copy of the index header for the opened index file handle to disk. The actual write occurs only if the header has been changed. This ensures that the index header on disk matches exactly the index header that is being maintained by BULLET.

Normally, this routine is not called directly but rather is done automatically when you unlock the file (UNLOCK_XB). This routine does not write out the header if the current lock state is shared (SIP.flags, bit2=1); it returns an error if tried. Unlocking a full-lock performs a flush automatically, and so you may never need to explicitly call this routine. Also, when relocking from an exclusive full-lock to a shared full-lock, an automatic flush is performed.

Assume the following: An index file with 100 keys. Your program opens the index file and adds 1 key. Physically, there are 101 keys on disk. However, the header image of the index file on disk still reads 100 keys. This isn't a problem; BULLET uses its in-memory copy of the index header and the in-memory copy does read 101 keys. But, if there were a system failure after the key add, the disk image of the header would not get updated since the disk image is written only on a CLOSE_ or FLUSH_INDEX_HEADER_XB, or on EXIT_XB (and also prior to REINDEX_XB). After the system restarts, BULLET opens the file, reads the header and thinks that there are 100 keys. You lost a key. Now, if after that key was added, your program issues a FLUSH_INDEX_HEADER_XB, the header on disk is refreshed with the in-memory copy, keeping the two in sync. The routine updates the directory entry, keeping things neat there as well (file size). 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.

Note: You should have a full-lock on the file before using this routine.

COPY_INDEX_HEADER_XB

Uses COPYPACK

```
    IN                      OUT
CP.func                 CP.stat
CP.handle
CP.filenamePtr
```

Copy the index file structure of an open index file to another file.

This routine duplicates the structure of an existing index file without having to re-specify the information needed by CREATE_INDEX_XB. The resultant index file will be exactly like the source, including sort function and key expression. It contains 0 keys.

ZAP_INDEX_HEADER_XB

Uses HANDLEPACK

```
    IN                      OUT
HP.func                 HP.stat
HP.handle
```

Delete all keys from an index file.

This routine is similar to COPY_INDEX_HEADER_XB except for one major difference: All keys in the source file are physically deleted.

If you have an index file with 100 keys and issue ZAP_INDEX_HEADER_XB, all 100 keys will be physically deleted and the file truncated to 0 keys. REINDEX_XB can be used to rebuild the index file.

Since BULLET reindexes in place, the use of ZAP is not typically needed.

Mid-Level Data

- GET_DESCRIPTOR_XB
- GET_RECORD_XB
- ADD_RECORD_XB
- UPDATE_RECORD_XB
- DELETE_RECORD_XB
- UNDELETE_RECORD_XB
- DEBUMP_RECORD_XB
- PACK_RECORDS_XB

GET_DESCRIPTOR_XB

Uses DESCRIPTORPACK and FIELDDESCTYPE

```
    IN                       OUT
DP.func                  DP.stat
DP.handle                DP.fieldNumber
DP.fieldNumber           DP.fieldOffset
      -or-               DP.FD.fieldName
DP.FD.fieldName          DP.FD.fieldType
                         DP.FD.fieldLen
                         DP.FD.fieldDC
                         DP.FD.altFieldLength
```

Get the field descriptor information for a field by fieldname or by field position.

To get descriptor info by fieldname, set DP.fieldNumber=0 and set the DP.FD.fieldName member to the fieldname string. Fieldnames must be 0-terminated and 0-filled, and must be upper-case, with A-Z and _ valid fieldname characters. If the string matches a fieldname in the DBF descriptor area, that field's descriptor info is returned in DP.FD, (FD is FIELDDESCTYPE), and its position is returned in FD.fieldNumber and FD.fieldOffset.

To get descriptor info by field position (i.e., field number), set DP.fieldNumber to the field's position. The first is field #1. The "delete tag" field is not considered a field. If the position is valid (i.e., greater than 0 and not beyond the last field), that field's descriptor info is returned in DP.FD.

This routine lets you work with *unknown* DBF files — those created by another program. By reading each field descriptor, by number, from 1 to number of fields (SDP.noFields), you can generate a run-time layout of the DBF file. Alternatively, you can get input from your user for a fieldname, and locate the descriptor by name.

If you need to add or delete a field, be sure to reindex all related index files so that their key expressions can be re-evaluated. To do this, you need to create a new data file and build it as you build any other new data file. Then, copy record-by-record from the old DBF to the new, using the old record layout for reads, and the new record layout for writes. After this, reindex any index file related to the DBF file. The old DBF file can then be deleted.

If non-standard fields are used (i.e., non-char structure members to match non-ASCII data fields in your non-standard DBF), then be aware that your compiler more than likely will add padding to align on member-size boundaries. This will result in a mis-match between your compiler structure and your DBF structure (as described in fieldList[]). To prevent this, place #pragma pack(1) / #pragma pack() around your structures that BULLET uses. Consult your particular compiler for alternate methods if it does not support #pragma pack.

GET_RECORD_XB

Uses ACCESSPACK

```
    IN                          OUT
AP.func                     AP.stat
AP.handle                   *AP.recPtr
AP.recNo
AP.recPtr
```

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

The data buffer, pointed to by AP.recPtr, 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, each 25 characters, then the variable would be typed as:

```
struct rectype {
CHAR  tag;             /* The Xbase DBF delete tag (must be included) */
CHAR  lastName[25];    /* same length as first field's descriptor fieldLen */
CHAR  firstName[25];   /* same length as second field's descriptor fieldLen */
}; /* 51 */
struct rectype recbuff;
```

The first record is at AP.recNo=1. The last is SDP.records, determined by STAT_DATA_XB. The buffer must be at least as large as the record length (SDP.recordLength).

This method of accessing the data file does not use any indexing. Generally, this access method is not used except for special purposes (sequential processing where order is not required). The preferred method to access the data is by one of the keyed GET_XB routines.

If non-standard fields are used (i.e., non-char structure members to match non-ASCII data fields in your non-standard DBF), then be aware that your compiler more than likely will add padding to align on member-size boundaries. This will result in a mis-match between your compiler structure (rectype above) and your DBF structure (as described in fieldList[]). To prevent this, place #pragma pack(1) / #pragma pack() around your structures that BULLET uses. Consult your particular compiler for alternate methods if it does not support #pragma pack.

ADD_RECORD_XB

Uses ACCESSPACK

```
    IN                          OUT
AP.func                     AP.stat
AP.handle                   AP.recNo
AP.recPtr
```

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

This method of adding a record involves no indexing. It is typically used to build a data file *en masse*. Indexing is deferred until all records have been added, and then quickly indexed using REINDEX_XB.

Since ADD_RECORD_XB is extremely fast, if you have several thousand data records to be added at once, appending records with this routine and reindexing when all have been added using REINDEX_XB is often faster than using INSERT_XB for each record to add.

The record number assigned to the record appended is determined by BULLET, and that record number is returned in AP.recNo.

If non-standard fields are used (i.e., non-char structure members to match non-ASCII data fields in your non-standard DBF), then be aware that your compiler more than likely will adding padding to align on member-size boundaries. This will result in a mis-match between your compiler structure and your DBF structure (as described in fieldList[]). To prevent this, place #pragma pack(1) / #pragma pack() around your structures that BULLET uses. Consult your particular compiler for alternate methods if it does not support #pragma pack.

UPDATE_RECORD_XB

Uses ACCESSPACK

IN	OUT
`AP.func`	`AP.stat`
`AP.handle`	
`AP.recNo`	
`AP.recPtr`	

Write the updated data record to the physical record number.

This method of updating a data record *must not* be used if any field being used as a key field (i.e., part of the key expression) is 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. You can then change it, and write the update out to disk using this routine.

If you need to change a field that is used as a key field, or part of one (e.g., SUBSTR()), use the UPDATE_XB routine.

If you plan on reindexing with REINDEX_XB immediately after using this routine, you may elect to update the data file using this method even if changing any field used as a key, rather than UPDATE_XB. This since UPDATE_XB is very disk intensive. However, if transaction support is needed (i.e., updates are dependent on other updates), then UPDATE_XB should be used.

DELETE_RECORD_XB

Uses ACCESSPACK

```
   IN                       OUT
AP.func                  AP.stat
AP.handle
AP.recNo
```

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 tag field. 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. 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 PACK_RECORDS_XB.

Records can be temporarily marked as deleted during processing and then recalled to normal status when completed, useful for flagging a record as having been processed (for example, mass updating using UPDATE_XB). The GET_XB routines return the record number associated with a key (in AP.recNo), and that record number can be used for this routine.

While the DELETE_RECORD_XB and UNDELETE_RECORD_XB routines provided in BULLET use the * and SPACE characters only, you can use whatever character you want in the tag field when you fill your record buffer structure's data. Normally, you set the tag field to SPACE (x.tag = ' ';), but, for example, if you want to implement your own, program-level locking you can use the tag field as a marker to indicate the record is locked (by using an 'L' character, or ID with bit7=1, or whatever you can think of) and use the very fast UPDATE_RECORD_XB to set it. Another possibility is set to aside a field to be used as this, say, along with the user ID of the lock owner.

The SKIP_TAG_SELECT item in SET_SYSVARS_XB can be set to have the REINDEX_XB routine not place a key value into the index file if a record has a matching tag field. This may be useful if you want to, say, generate an *ad hoc* index for only undeleted records.

UNDELETE_RECORD_XB

Uses ACCESSPACK

```
   IN                       OUT
AP.func                  AP.stat
AP.handle
AP.recNo
```

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. This routine removes the * character, as put there by DELETE_RECORD_XB, in the tag field and replaces it with a SPACE. The tag field is always overwritten with a SPACE, regardless of what it was.

DEBUMP_RECORD_XB

Uses ACCESSPACK

```
   IN                      OUT
AP.func                 AP.stat
AP.handle
AP.recNo
```

Remove the record identified by AP.recNo from the data file if and only if the record is the last in the file. The file is automatically flushed before debumping.

Unlike DELETE_RECORD_XB, this routine physically removes a data record from the DBF file, provided that the record to delete is the last. STAT_DATA_XB can be used to identify the last record number (SDP.records is the last). This, when used after deleting any and all keys in all index files referencing this record (see DELETE_KEY_XB), is useful if you are managing a transaction log and need to back out changes made, beyond what BULLET performs.

If the record is not the last, alternate methods must be used. The simplest, and often equally as good as physically deleting the record, is to just mark the record as deleted using DELETE_RECORD_XB and let it remain in the file until the next PACK_RECORDS_XB. Another option is to overwrite the record's data with SPACES, or other appropriate field data (such as HIGH-VALUES, and use UPDATE_XB), if necessary. This routine is the only method available to physically remove a record from the file, short of using PACK_RECORDS_XB.

Removing a record with active keys referencing that record will result in an access error (ERR_UNEXPECTED_EOF) when accessing that key with GET_XB routines, or will generate stale results. Remove any keys that reference this record before deleting it.

PACK_RECORDS_XB

Uses ACCESSPACK

```
   IN                      OUT
AP.func                 AP.stat
AP.handle
```

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

Packing occurs in place using the existing file. It is **recommended** that you use BACKUP_FILE_XB 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. All records with the tag field set to * are removed from the file. If there are index files for this DBF file, they <u>must</u> be reindexed after the pack process by using REINDEX_XB.

Memo files are not affected by this routine. Before packing, it is recommended that you traverse the data file to be packed, and for records that are to be deleted, check to see if there is a memo record. If there is, delete the memo. Do this for each such occurrence. This way, orphaned memo records will not take up permanent space in the memo file.

Mid-Level Memo

- GET_MEMO_SIZE_XB
- GET_MEMO_XB
- ADD_MEMO_XB
- UPDATE_MEMO_XB
- DELETE_MEMO_XB
- MEMO_BYPASS_XB

GET_MEMO_SIZE_XB

Uses MEMODATAPACK

IN	OUT
MDP.func	MDP.stat
MDP.dbfHandle	MDP.memoBytes
MDP.memoNo	

Get the number of bytes used by the memo at MDP.memoNo.

Memo file allocation is made in blocks, typically of 512 bytes each. Therefore, a memo of 10 bytes uses 1 allocation block, as would a 500-byte memo. This size is stored with each memo record, and can be retrieved. Before accessing a memo record, it's a good idea to retrieve the current size of the memo so you know how large a buffer you may need if you intend to read it all in, at one time, or even to just know how much to read, in total, reading parts of it at a time.

The first memo is at MDP.memoNo=1. The last memo number cannot be easily determined, but generally this does not need to be known. The memo number identifying the memo's location is stored in the memo field area of the DBF record. It is stored as a text string (e.g., "0000000001"). This is not a C string; there is no zero terminator so sprintf() should be used. This number is the physical block number at which the memo starts. Memos are always stored in consecutive blocks, if more than a single block is needed. For example, a memo of 513 bytes uses two blocks, say, #1 and #2. The next memo added would use memo #3 (if #3 is available), rather than #2 since #2 was used by the first memo. Memo numbers may be reassigned (see UPDATE_MEMO_XB). The highest possible memo number is 589,822 (0x8FFFE). With the standard 512-byte block size, this allows a memo file to be up to 288MB. If more memo data space is needed, use a larger block size (e.g., 2KB block size allows over 1GB per memo file).

Notice For All Memo Routines

In multitasking environments you should have a full-lock on the DBF file that owns this memo file, or at least a record lock on the record that owns the memo number. In BULLET, locking is not performed on the memo file. Instead, the lock is implied when the lock is made on the DBF file. This because a memo file is for one DBF file alone, and so if you have a lock on the DBF before accessing the memo file (for whatever reason), then no other process may lock the DBF and also access the memo.

This works only if you restrict your access to the memo file if you have a lock on the DBF master file (the DBF that this DBT memo file belongs to) or on the DBF record. For this routine, which only requires access to this memo record, a record lock is sufficient since no writing is performed. Further, a shared lock is all that is required. This because all that is required to keep from stepping on other process's toes is that it be known that the current memo header info (for this memo record), as known to this process, is the current state of this memo. In other words, it must be true that the memo file state on disk exactly matches the memo file state in memory. With a lock in place, no other process may gain write access to change this memo, "out from under you". A shared lock does allow the other process to read this memo, and that may be used if no writing is needed.

Each memo routine following states its lock requirements (exclusive full lock, shared full lock, exclusive record lock, or shared record lock).

GET_MEMO_XB

Uses MEMODATAPACK

IN	OUT
`MDP.func`	`MDP.stat`
`MDP.dbfHandle`	`*MDP.memoPtr`
`MDP.memoNo`	`MDP.memoBytes`
`MDP.memoPtr`	
`MDP.memoOffset`	
`MDP.memoBytes`	

Read the specified number of bytes of the memo, starting at the offset, into the buffer. The actual number of bytes read is returned.

Use GET_MEMO_SIZE_XB to determine that total number of bytes you may need to read. With that, you can allocate a buffer of that size to read the entire memo into. Or, you can read chunks of the memo, a chunk at a time, up to the number of bytes in the memo.

The number of bytes actually read (and stored starting at MDP.memoPtr) is returned in MDP.memoBytes (overwriting the value you placed there). If the number of bytes requested is not the same as the number of bytes returned, you attempted to read beyond the end of the memo. BULLET does *not* return an error if you try this, which is SOP for file reads, so check the two if you need to verify this. An error is returned, however, if you attempt to read at a starting offset beyond the end of the actual memo data (i.e., MDP.memoOffset > memo's data size). The first byte of the memo data is at .memoOffset=0.

It's recommended that a lock be in force on either the DBF (full-lock) or on the record that this memo belongs to. A shared lock is okay since no writing is done.

ADD_MEMO_XB

Uses MEMODATAPACK

IN	OUT
`MDP.func`	`MDP.stat`
`MDP.dbfHandle`	`MDP.memoNo`
`MDP.memoPtr`	
`MDP.memoBytes`	

Add the data from the buffer for the specified bytes to a new memo. The memo number used is returned.

Any data can be stored in a memo. The memo number returned can be any value; it can even be less than the previous add's memo number. The reason for this is that an avail-list is kept for the memo file, and any deleted or otherwise freed blocks become available for re-use. The memo is stored in the first contiguous group of free blocks large enough to satisfy the request. For example, if MDP.memoBytes is from 1 to (blockSize-8) bytes, the first available block is used. If the size needed is greater than 1 block, then the avail-list is walked and the first contiguous group large enough to satisfy the request is used. If none of the avail-list groups is large enough, ultimately, the new memo data is appended to the end of the file. This is also done if there are no avail-list items at all, such as in a memo file that has never had deletes or updates.

The returned memo is a binary block number (ULONG). This value should be converted into an ASCII string (sprintf can be used) and stored in the DBF data record, in the memo field. The string should be of the form, "0000000001" (for MDP.memoNo=1), with leading zeros, but no zero terminator (exactly 10 bytes in size). This data record should then be written to disk using UPDATE_RECORD_XB.

Since BULLET can be used in non-standard Xbase mode, where binary field values can be used, you can omit the conversion from binary to ASCII if a standard DBF is not required. Likewise, when accessing a memo, the conversion of the memo block number from ASCII to binary would not be required.

It's recommended that a lock be in force on the DBF (full-lock). A shared lock may not be used since writing to the memo file, and the DBF record, is required. A full lock is required since the memo file header is read and written.

UPDATE_MEMO_XB

Uses MEMODATAPACK

IN	OUT
`MDP.func`	`MDP.stat`
`MDP.dbfHandle`	`MDP.memoNo`
`MDP.memoNo`	
`MDP.memoPtr`	
`MDP.memoOffset`	
`MDP.memoBytes`	

Update an existing memo. The update can overwrite current data, append new data extending the current size, or it can shrink the current size.

Appending data so that the memo is extended may result in a new memo number returned. The original memo blocks are made available for reuse (deleted). Shrinking will not change the memo number, but unused blocks from the shrink are made available for reuse.

If you want to change anything in the memo at MDP.memoNo, locate its position within the memo with MDP.memoOffset and set the size in MDP.memoBytes. The first data byte of a memo is located at MDP.offset=0. There are 8 bytes of overhead per memo record (any number of blocks still has only the 8 bytes of overhead), but these are transparent to any memo access you do. The bytes at MDP.memoPtr overwrite the current memo data at the position specified. For example, if you want to change the first 5 bytes of the first memo, set MDP.memoNo=1, MDP.memoPtr=yourNewData, MDP.memoOffset=0, and MDP.memoBytes=5. On return, MDP.memoNo is going to be the same as it was before the update, since you are not extending the memo size in this example. Nothing further needs to be done; the memo is updated.

If you want to add new memo data to an existing memo at MDP.memoNo, such as adding another line item, or problem report paragraph, etc., set MDP.memoOffset=theCurrentMemoSize (this locates to the end of the current memo data), MDP.memoBytes=bytesYouWantToAppend, and MDP.memoPtr = yourDataToAppend. If the old data size plus your newly added data still fits inside the last memo block previously used, MDP.memoNo is returned the same as it was on entry. However, if the new data requires that more blocks be allocated, the entire memo is relocated to the next contiguous block group that is large enough to store the data. That new block number is returned in MDP.memoNo, and the old block number and all its blocks are placed on the top of the avail-list.

If you want to shrink the size as reported by GET_MEMO_SIZE_XB from an existing memo at MDP.memoNo, set MDP.memoBytes=newSizeYouWant, and MDP.memoPtr=NULL. This means that you should have, before making this shrink call, updated the memo data that occurs within this new size to be the data size you want to be in the memo. For example, if you have 10 line items, say, each 60 bytes long, and want to remove line item #5, you could do it by reading all 10 line items to memory, moving line items #6 to 10 down one (so they are now line items #5 to 9, effectively removing old line item #5), and update the memo (by using memoOffset=0 and memoBytes=9*60). After this, though, you still have 10*60 bytes as the memo size (old line item #10 is now at #9 and still at #10). Since you want the size to reflect the real data in the memo, set MDP.memoBytes=90, MDP.memoPtr = NULL, and update this memo number. Only the memo's size is affected by this particular update. The size specified must be smaller than the original size, or an error is returned.

It's recommended that a lock be in force on the DBF (full-lock). A record lock should not be used if the update may result in blocks being moved, or the memo being shrunk by a full block or more. A shared lock may not be used since writing to the memo file, and to the DBF record if MDP.memoNo is new, is required.

DELETE_MEMO_XB

Uses MEMODATAPACK

IN	OUT
`MDP.func`	`MDP.stat`
`MDP.dbfHandle`	
`MDP.memoNo`	

The memo and all its blocks are made available for reuse.

Before using PACK_RECORDS_XB, you should run through all DBF records and check for those records that are deleted (record.tag='*') to be sure that any memo belong to those records are deleted from the memo file. If this is not done, orphaned memo records — those that do not have a DBF record memo field pointing to it, may be left in the memo file (forever!).

After deleting a memo record, update the DBF record's memo field by writing <SPACES> (ASCII 32) to the memo field member. Update this to disk with UPDATE_RECORD_XB as soon as possible (and before unlocking). A memo field with no current memo record is indicated by spaces ("0000000000" should not be used).

It's recommended that a lock be in force on the DBF (full-lock). Neither a record lock nor a shared lock may be used since writing to the memo file header and the DBF record is required.

MEMO_BYPASS_XB

Uses MEMODATAPACK

```
   IN                     OUT
MDP.func               MDP.stat
MDP.dbfHandle
MDP.memoBypass
```

Memo files are created, opened, closed, and flushed/reloaded by their corresponding DBF data file action. To perform these tasks asynchronously, this routine is used. Bypass routines are:

MDP.memoBypass	Value
BYPASS_CREATE_MEMO	1
BYPASS_OPEN_MEMO	2
BYPASS_CLOSE_MEMO	3
BYPASS_READ_MEMO_HEADER	4
BYPASS_FLUSH_MEMO_HEADER	5

All bypass routines require the handle of the DBF file that this memo is for. Nothing is returned here, except the result code. The memo handle from the open is stored internally, but is available by using STAT_DATA_XB and checking SDP.memoHandle. However, none of the BULLET memo routines use the memo handle directly; all access to the memo file is through the master DBF file handle.

No data is required for input other than the DBF handle and memo bypass routine to perform (see table above). All required info is obtained from the DBF file's information. You may use an alternate block size, as set via SET_SYSVARS_XB.

Generally, there is no need to call these routines using this bypass. However, if you need to create a memo file anew (say, after the initial DBF was created), and then open it, using these routines is the easiest way to proceed.

Note: When creating a memo via the bypass method, the **file ID is altered** to indicate that the DBF has a DBT memo file. The file ID is the first byte of the DBF file. The ID is changed by OR'ing 0x88h with the current file ID value. The next flush or close updates the disk image of the DBF with the new file ID. The next DBF open, then, also opens the DBT memo file created here. Be sure to always keep the DBT and DBF pairs in the same directory, if moved.

Since the DBF file is already open (and must be to use any of these routines), you must use the open bypass routine to open the memo if you plan on using it. Either that, or close the DBF after you've create the memo file, and simply re-open the DBF, which also, now, opens the DBT memo file.

The other available bypass routines: close, read, and flush, typically will not be used from this bypass routine. These operations are done automatically when their corresponding DBF action is performed, and have little functionality used on their own.

Before using BYPASS_READ_MEMO_HEADER or BYPASS_FLUSH_MEMO_HEADER, it's recommended that a lock be in force on the DBF (full-lock). A shared lock can be used for BYPASS_READ_MEMO_HEADER, but it must be a full lock.

Mid-Level Index

- FIRST_KEY_XB
- EQUAL_KEY_XB
- EQUAL_OR_GREATER_KEY_XB
- EQUAL_OR_LESSER_KEY_XB
- NEXT_KEY_XB
- PREV_KEY_XB
- LAST_KEY_XB
- STORE_KEY_XB
- DELETE_KEY_XB
- BUILD_KEY_XB
- GET_CURRENT_KEY_XB
- GET_KEY_FOR_RECORD_XB

FIRST_KEY_XB

Uses ACCESSPACK

IN	OUT
AP.func	AP.stat
AP.handle	AP.recNo
AP.keyPtr	*AP.keyPtr

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 logical key of the index file, returning it, and also returning the record number within the DBF that the key indexes.

To retrieve the data record you can use the GET_RECORD_XB routine. The preferred method, however, is to use GET_FIRST_XB, which combines these operations.

The key returned includes an enumerator if the index file allows duplicate keys. 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 NEXT_KEY_XB.

If an external data file was specified in CREATE_INDEX_XB, the record number returned by this routine does not refer to a DBF record, but rather is the value supplied when the key was stored. This permits index access to your data files (data files which are not maintained by BULLET, but by you).

EQUAL_KEY_XB

Uses ACCESSPACK

IN	OUT
`AP.func`	`AP.stat`
`AP.handle`	`AP.recNo`
`AP.keyPtr`	

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 the enumerator word if the index file is using non-unique keys.

To retrieve the data record you can use the GET_RECORD_XB routine. The preferred method, however, is to use GET_EQUAL_XB, which combines these operations.

This routine returns no key in *keyPtr since, by definition, you already have the key in the key buffer if this routine succeeds.

This routine finds only an *exact* match to the specified key (including the enumerator if applicable). However, even if the exact key is not found, the index file is positioned so that a NEXT_KEY_XB retrieves the key that would have followed the unmatched specified key. For example, if the key to match were "KINGS" (a partial key, say, with \0\0 after the S), EQUAL_KEY_XB would return a key not found error (since no exact match was found). If you were to now do a NEXT_KEY_XB, the next key logically ordered after "KINGS" would be returned. Let's say "KINGSTON" was the next. That key value, including enumerator if any, and the key's record number is returned from the NEXT_KEY_XB call. This technique lets you position anywhere in the index file to narrow down any manual searches (for instance, if you're looking for a key but aren't sure of the exact spelling).

Note: When using the partial key technique shown above, be sure to set the unspecified characters of the key to \0, or at least the two bytes immediately following your search criterion. This for both unique and non-unique index files. This is to ensure that the key located is the **first** key matching your search criterion.

EQUAL_OR_GREATER_KEY_XB

Uses ACCESSPACK

IN	OUT
`AP.func`	`AP.stat`
`AP.handle`	`AP.recNo`
`AP.keyPtr`	`*AP.keyPtr`

Search for the exact key in the index file and, if not found, get the key that would have followed it.

This routine is similar to EQUAL_KEY_XB except that this routine returns a key in *keyPtr (either the same as on entry, or if that is not found, then the next greater key).

The main benefit of this routine is that it is an atomic operation. It differs from setting the atomic mode flag of SET_SYSVARS_XB in that this routine allows a fuzzy starting point: the key in AP.keyPtr, on entry (IN), need not exist.

EQUAL_OR_LESSER_KEY_XB

Uses ACCESSPACK

```
    IN                       OUT
AP.func                 AP.stat
AP.handle               AP.recNo
AP.keyPtr               *AP.keyPtr
```

Search for the exact key in the index file and, if not found, get the key that would have come before it.

This routine is similar to EQUAL_KEY_XB except that this routine returns a key in *keyPtr (either the same as on entry, or if that is not found, then the previous, lesser key.

The main benefit of this routine is that it is an atomic operation. It differs from setting the atomic mode flag of SET_SYSVARS_XB in that this routine allows a fuzzy starting point: the key in AP.keyPtr, on entry (IN), need not exist.

NEXT_KEY_XB

Uses ACCESSPACK

```
    IN                       OUT
AP.func                 AP.stat
AP.handle               AP.recNo
AP.keyPtr               *AP.keyPtr
```

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 retrieve 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 GET_RECORD_XB routine. The preferred method, however, is to use GET_NEXT_XB, which combines these operations.

The key returned includes an enumerator if the index file allows duplicates.

This routine is typically called after the index file has first been positioned to a known key using either FIRST_KEY_XB or EQUAL_KEY_XB, or after a previous NEXT_KEY_XB or even PREV_KEY_XB. What it basically does is get the key following the current key, and then makes that key the new current key.

If bit0 of the atomic mode flag of SET_SYSVARS_XB is set to 1, key access is based on a given starting point. This simplifies index access in multi-threaded code, where another thread may have altered the last key accessed in the index file. This mode lets you set a starting point for the operation by supplying in AP.keyPtr the key value to start at.

For example, say you use GET_FIRST_XB. On return, AP.keyPtr has the the very first key. Say elsewhere in your multi-threaded program, another operation accesses that same index file handle, and performs some other access, where the last accessed key is no longer the same (i.e., not the first key). Your first thread is expecting that a GET_NEXT_XB would get the second key, however, it very likely won't since the second thread has altered the last accessed key for that file handle. By using the atomic mode for key access, your first thread, which has the first key value in its AP.keyPtr, can do a call to GET_NEXT_XB and get expected results, since the NEXT operation first positions to the value in AP.keyPtr and then follows up with a GET_NEXT operation. This is performed within the Bullet kernel, and so won't be interrupted by another thread (i.e., it is an atomic operation). For this to work, you must ensure that the AP.keyPtr value is set to the value of the last accessed key. This will always be the case unless uninitialized, or you are using global variables for your threads' AP (AccessPack). On return from the operation, AP.keyPtr will once again be set up for another atomic operation.

Note: You must supply a valid key value for this atomic access mode. AP.keyPtr must be at least as large as the key length in all cases, and is to have the starting point for the operation (i.e., the last accessed key). You may, alternatively, set the first byte of the key buffer to 0 (but not AP.keyPtr itself to NULL). This disables atomic mode for that access, and reverts to the internally-stored last key accessed as the starting point.

PREV_KEY_XB

Uses ACCESSPACK

IN	OUT
`AP.func`	`AP.stat`
`AP.handle`	`AP.recNo`
`AP.keyPtr`	`*AP.keyPtr`

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 retrieve 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 GET_RECORD_XB routine. The preferred method, however, is to use GET_PREV_XB, which combines these operations.

The key returned includes an enumerator if the index file allows duplicates.

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

If bit0 of the atomic mode flag of SET_SYSVARS_XB is set to 1, key access is based on a given starting point. This simplifies index access in multi-threaded code, where another thread may have altered the last key accessed in the index file. This mode lets you set a starting point for the operation by supplying in AP.keyPtr the key value to start at.

For example, say you use GET_FIRST_XB. On return, AP.keyPtr has the the very first key. Say elsewhere in your multi-threaded program, another operation accesses that same index file handle, and performs some other access, where the last accessed key is no longer the same (i.e., not the first key). Your first thread is expecting that a GET_NEXT_XB would get the second key, however, it very likely won't since the second thread has altered the last accessed key for that file handle. By using the atomic mode for key access, your first thread, which has the first key value in its AP.keyPtr, can do a call to GET_NEXT_XB and get expected results, since the NEXT operation first positions to the value in AP.keyPtr and then follows up with a GET_NEXT operation. This is performed within the Bullet kernel, and so won't be interrupted by another thread (i.e., it is an atomic operation). For this to work, you must ensure that the AP.keyPtr value is set to the value of the last accessed key. This will always be the case unless uninitialized, or you are using global variables for your threads' AP (AccessPack). On return from the operation, AP.keyPtr will once again be set up for another atomic operation.

Note: You must supply a valid key value for this atomic access mode. AP.keyPtr must be at least as large as the key length in all cases, and is to have the starting point for the operation (i.e., the last accessed key). You may, alternatively, set the first byte of the key buffer to 0 (but not AP.keyPtr itself to NULL). This disables atomic mode for that access, and reverts to the internally-stored last key accessed as the starting point.

LAST_KEY_XB

Uses ACCESSPACK

IN	OUT
`AP.func`	`AP.stat`
`AP.handle`	`AP.recNo`
`AP.keyPtr`	`*AP.keyPtr`

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 GET_RECORD_XB routine. The preferred method, however, is to use GET_LAST_XB, which combines these operations.

The key returned includes an enumerator if the index file allows duplicates.

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 PREV_KEY_XB.

STORE_KEY_XB

Uses ACCESSPACK

IN	OUT
`AP.func`	`AP.stat`
`AP.handle`	
`AP.recNo`	
`AP.keyPtr`	

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 INSERT_XB instead.

To do a complete data record and key insert, use ADD_RECORD_XB to add the data record to the DBF, BUILD_KEY_XB to construct the key, then STORE_KEY_XB to insert the key and record number information into the index file. If that key already exists and the file allows duplicate keys, attach the proper enumerator word and retry STORE_KEY_XB. INSERT_XB does this automatically.

DELETE_KEY_XB

Uses ACCESSPACK

```
    IN                      OUT
AP.func                 AP.stat
AP.handle
AP.keyPtr
```

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 duplicate keys are allowed.

Typically, this routine would seldom be used since deleted DBF data records are only physically deleted during a PACK_RECORDS_XB operation, after which REINDEX_XB is done. It is useful if you are managing a transaction log and need to back out changes made, beyond what BULLET performs. Also see DEBUMP_RECORD_XB.

If you have non-unique keys (where DUPS_ALLOWED is true), you may have several keys that match your criterion, and only differ in their enumerator. To identify which key, then, goes to a particular DBF record, compare that key's AP.recNo with the number of your DBF record. If they are the same, then this key belongs to that record. Use either the KEY_XB or the GET_XB routines, then, before using this routine. In other words, use this routine only after you have identified *exactly* the key to delete, and for the exact data record. Once you have the record number, you can locate its key by using GET_KEY_FOR_RECORD_XB.

BUILD_KEY_XB

Uses ACCESSPACK

```
    IN                      OUT
AP.func                 AP.stat
AP.handle               *AP.keyPtr
AP.recPtr
AP.keyPtr
```

Build the key for the specified data record based on the key expression for the index file. If the index file allows duplicate keys, a 0-value enumerator is attached.

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. If used, it normally would be used prior to STORE_KEY_XB.

This routine can be replaced. See *Custom Build-Key Routine*.

Note: If DUPS_ALLOWED, this routine always sets the enumerator to \0\0. Enumerator management, which is used to guarantee a unique key, is performed only when the INSERT_XB routine is used.

GET_CURRENT_KEY_XB

Uses ACCESSPACK

IN	OUT
`AP.func`	`AP.stat`
`AP.handle`	`AP.recNo`
`AP.keyPtr`	`*AP.keyPtr`

Retrieve the current key value for the specified index file handle and also the data record number that it indexes. The key value includes the enumerator if applicable.

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 associated with that key). STAT_INDEX_XB returns this information, too.

GET_KEY_FOR_RECORD_XB

Uses ACCESSPACK

IN	OUT
`AP.func`	`AP.stat`
`AP.handle`	`*AP.keyPtr`
`AP.recNo`	
`AP.recPtr`	
`AP.keyPtr`	

Retrieve the key for the record/record number pair.

This routine would typically be used prior to using DELETE_KEY_XB and DEBUMP_RECORD_XB. The key returned includes the enumerator if applicable.

This routine sifts through any duplicate keys (if DUPS_ALLOWED) for the key that matches the record/record number pair, and so requires both the actual data record along with its physical record number (even if dups are not allowed).

Typically this routine is extraneous; the key is available with a GET_XB routine and so can be deleted from the information provided through normal access.

This routine builds a key based on the supplied record at AP.recPtr and searches the index for that key proper. If found, and if DUPS_ALLOWED, each key matching the key proper has its record number compared to the record number in AP.recNo. If that matches, too, then that is the exact key being sought.

High-Level Index+Data

- GET_FIRST_XB
- GET_EQUAL_XB
- GET_EQUAL_OR_GREATER_XB
- GET_EQUAL_OR_LESSER_XB
- GET_NEXT_XB
- GET_PREV_XB
- GET_LAST_XB
- INSERT_XB
- UPDATE_XB
- REINDEX_XB

GET_FIRST_XB

Uses ACCESSPACK

IN	OUT
AP.func	AP.stat
AP.handle	AP.recNo
AP.recPtr	*AP.recPtr
AP.keyPtr	*AP.keyPtr

Retrieve the first indexed key and its data record.

The key returned includes an enumerator if the index file uses non-unique keys (DUPS_ALLOWED).

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 GET_NEXT_XB, until the end of the database is reached, at which point an error is returned.

This routine, like all the high-level GET_XB routines, fills in the AP.recNo of the record accessed. In this case, it fills AP.recNo with the record number pointed to by the first key. Since this is done upon each GET_XB access, the AP pack is primed for an UPDATE_XB

GET_EQUAL_XB

Uses ACCESSPACK

IN	OUT
AP.func	AP.stat
AP.handle	AP.recNo
AP.recPtr	*AP.recPtr
AP.keyPtr	

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

This routine finds only an *exact* match to the specified key (including the enumerator if applicable).

This routine, like all the high-level GET_XB routines, fills in the AP.recNo of the record accessed. In this case, it fills AP.recNo with the record number pointed to by the matching key. Since this is done upon each GET_XB access, the AP pack is primed for an UPDATE_XB

GET_EQUAL_OR_GREATER_XB

Uses ACCESSPACK

IN	OUT
`AP.func`	`AP.stat`
`AP.handle`	`AP.recNo`
`AP.recPtr`	`*AP.recPtr`
`AP.keyPtr`	`*AP.keyPtr`

Search for the exact key in the index file and return its data record, or if not found, get the key and record that would have followed it.

This routine is similar to GET_EQUAL_XB except that this routine returns a key in *keyPtr (either the same as on entry, or if that is not found, then the next greater key).

The main benefit of this routine is that it is an atomic operation. It differs from setting the atomic mode flag of SET_SYSVARS_XB in that this routine allows a fuzzy starting point: the key in AP.keyPtr, on entry (IN), need not exist.

GET_EQUAL_OR_LESSER_XB

Uses ACCESSPACK

IN	OUT
`AP.func`	`AP.stat`
`AP.handle`	`AP.recNo`
`AP.recPtr`	`*AP.recPtr`
`AP.keyPtr`	`*AP.keyPtr`

Search for the exact key in the index file and return its data record, or if not found, get the key and record that would have come before it.

This routine is similar to GET_EQUAL_XB except that this routine returns a key in *keyPtr (either the same as on entry, or if that is not found, then the previous, lesser key).

The main benefit of this routine is that it is an atomic operation. It differs from setting the atomic mode flag of SET_SYSVARS_XB in that this routine allows a fuzzy starting point: the key in AP.keyPtr, on entry (IN), need not exist.

GET_NEXT_XB

Uses ACCESSPACK

IN	OUT
`AP.func`	`AP.stat`
`AP.handle`	`AP.recNo`
`AP.recPtr`	`*AP.recPtr`
`AP.keyPtr`	`*AP.keyPtr`

Retrieve the next indexed key and its data record.

The key returned includes an enumerator if the index file uses non-unique keys (DUPS_ALLOWED).

This routine is typically called after the index file has first been positioned to a known key using either GET_FIRST_XB or GET_EQUAL_XB, or after a previous GET_NEXT_XB or even GET_PREV_XB. What it basically does is get the key and data record following the current key, and then makes that key the new current key.

This routine, like all the high-level GET_XB routines, fills in the AP.recNo of the record accessed. In this case, it fills AP.recNo with the record number pointed to by the next key (now the current key). Since this is done upon each GET_XB access, the AP pack is primed for an UPDATE_XB.

If bit0 of the atomic mode flag of SET_SYSVARS_XB is set to 1, key access is based on a given starting point. This simplifies index access in multi-threaded code, where another thread may have altered the last key accessed in the index file. This mode lets you set a starting point for the operation by supplying in AP.keyPtr the key value to start at.

For example, say you use GET_FIRST_XB. On return, AP.keyPtr has the the very first key. Say elsewhere in your multi-threaded program, another operation accesses that same index file handle, and performs some other access, where the last accessed key is no longer the same (i.e., not the first key). Your first thread is expecting that a GET_NEXT_XB would get the second key, however, it very likely won't since the second thread has altered the last accessed key for that file handle. By using the atomic mode for key access, your first thread, which has the first key value in its AP.keyPtr, can do a call to GET_NEXT_XB and get expected results, since the NEXT operation first positions to the value in AP.keyPtr and then follows up with a GET_NEXT operation. This is performed within the Bullet kernel, and so won't be interrupted by another thread (i.e., it is an atomic operation). For this to work, you must ensure that the AP.keyPtr value is set to the value of the last accessed key. This will always be the case unless uninitialized, or you are using global variables for your threads' AP (AccessPack). On return from the operation, AP.keyPtr will once again be set up for another atomic operation.

Note: You must supply a valid key value for this atomic access mode. AP.keyPtr must be at least as large as the key length in all cases, and is to have the starting point for the operation (i.e., the last accessed key). You may, alternatively, set the first byte of the key buffer to 0 (but not AP.keyPtr itself to NULL). This disables atomic mode for that access, and reverts to the internally-stored last key accessed as the starting point.

GET_PREV_XB

Uses ACCESSPACK

IN	OUT
`AP.func`	`AP.stat`
`AP.handle`	`AP.recNo`
`AP.recPtr`	`*AP.recPtr`
`AP.keyPtr`	`*AP.keyPtr`

Retrieve the previous indexed key and its data record.

The key returned includes an enumerator if the index file uses non-unique keys (DUPS_ALLOWED).

This routine is typically called after the index file has first been positioned to a known key using either GET_LAST_XB or GET_EQUAL_XB, or after a previous GET_PREV_XB or even GET_NEXT_XB. What it basically does is get the key and data record preceding the current key, and then makes that key the new current key.

This routine, like all the high-level GET_XB routines, fills in the AP.recNo of the record accessed. In this case, it fills AP.recNo with the record number pointed to by the previous key (now the current key). Since this is done upon each GET_XB access, the AP pack is primed for an UPDATE_XB.

If bit0 of the atomic mode flag of SET_SYSVARS_XB is set to 1, key access is based on a given starting point. This simplifies index access in multi-threaded code, where another thread may have altered the last key accessed in the index file. This mode lets you set a starting point for the operation by supplying in AP.keyPtr the key value to start at.

For example, say you use GET_FIRST_XB. On return, AP.keyPtr has the the very first key. Say elsewhere in your multi-threaded program, another operation accesses that same index file handle, and performs some other access, where the last accessed key is no longer the same (i.e., not the first key). Your first thread is expecting that a GET_NEXT_XB would get the second key, however, it very likely won't since the second thread has altered the last accessed key for that file handle. By using the atomic mode for key access, your first thread, which has the first key value in its AP.keyPtr, can do a call to GET_NEXT_XB and get expected results, since the NEXT operation first positions to the value in AP.keyPtr and then follows up with a GET_NEXT operation. This is performed within the Bullet kernel, and so won't be interrupted by another thread (i.e., it is an atomic operation). For this to work, you must ensure that the AP.keyPtr value is set to the value of the last accessed key. This will always be the case unless uninitialized, or you are using global variables for your threads' AP (AccessPack). On return from the operation, AP.keyPtr will once again be set up for another atomic operation.

Note: You must supply a valid key value for this atomic access mode. AP.keyPtr must be at least as large as the key length in all cases, and is to have the starting point for the operation (i.e., the last accessed key). You may, alternatively, set the first byte of the key buffer to 0 (but not AP.keyPtr itself to NULL). This disables atomic mode for that access, and reverts to the internally-stored last key accessed as the starting point.

GET_LAST_XB

Uses ACCESSPACK

IN	OUT
`AP.func`	`AP.stat`
`AP.handle`	`AP.recNo`
`AP.recPtr`	`*AP.recPtr`
`AP.keyPtr`	`*AP.keyPtr`

Retrieve the last indexed key and its data record.

The key returned includes an enumerator if the index file uses non-unique keys (DUPS_ALLOWED).

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 GET_PREV_XB, until the top of the database is reached, at which point an error is returned.

This routine, like all the high-level GET_XB routines, fills in the AP.recNo of the record accessed. In this case, it fills AP.recNo with the record number pointed to by the last key. Since this is done upon each GET_XB access, the AP pack is primed for an UPDATE_XB

INSERT_XB

Uses ACCESSPACK

IN	OUT
`AP.func`	`AP.stat`
`AP.handle`	`AP.recNo`
`AP.recNo`	`*AP.keyPtr`
`AP.recPtr`	
`AP.keyPtr`	
`AP.nextPtr`	

Append the data records to data files and build and insert the related keys into all linked index files. (Alternate forms are possible.)

This routine is used to add new entries into a database. Up to 256 index files may be inserted into per call, with up to 256 data files being added, too, for a total of 512 files managed per single INSERT_XB call.

Note: Bullet comes in 100, 250, and 1024-file versions and so this routine is able to use as many files as handles are still available.

If non-standard fields are used (i.e., non-char structure members to match non-ASCII data fields in your non-standard DBF), then be aware that your compiler more than likely will add padding to align on member-size boundaries. This will result in a mis-match between your compiler structure and your DBF structure (as described in fieldList[]). To prevent this, place #pragma pack(1) / #pragma pack() around your structures that BULLET uses. Consult your particular compiler for alternate methods if it does not support #pragma pack.

Only index handles are listed in AP.handle. Each index file has associated with it a data file, known internally to BULLET (the xbLink from OPEN_XB). There may be more than one index file for a data file, but there is always one data file per index handle specified in the list. In other words, you can list five index files, each indexing the same xbLink data file, and have BULLET perform an **atomic** insert of that list. Or, another possibility is that you have a single index file, indexing a single data file. Or, you can list 256 index files, each indexing a single data file (512 total files).

This and several other routines are transaction-list-based. This means that 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 index files) effectively restored to its original state.

If the routine failed to complete, the function return value is the number (1-based) of the pack that caused the failure. A positive number indicates the failure was from an index operation; a negative number indicates the failure was from a data operation. In each case, the absolute value of the return code is the list item that failed (the pack index). For example, if five index handles are in the list (AP[0] to AP[4]), and an error occurred on the last pack's index file, the return code would be positive 5, indicating the fifth pack (AP[4]) failed. Since it was a positive 5, the index file was being processed when the error occurred. *Being processed* means not only physical access, but verification, etc. If the return code was -5, then again, the error was in the fifth pack, but since it is negative, the error occurred while processing the data file. In either case, upon return, the database is effectively restored to the way it was before the INSERT_XB call was made. Remedy the error, if possible, and INSERT_XB again.

Each pack **must** include a separate key buffer. You must not share a common key buffer. Doing so disables any chance of recovering the index files in case of error, since it is in these buffers that BULLET places the newly built keys, and it is from these that BULLET, upon an error condition, deletes the keys (required for roll-back).

The enumerator is automatically set up by this routine, if required (DUPS_ALLOWED and the key already exists with enumerator 0). It does this by seeking the last possible enumerator value (0xFFFF) and then backing up to the previous key. That key's enumerator is evaluated and incremented, and used as this key's.

Specifying Files

As mentioned, only the index file handles are specified in AP.handle. Data files are implicitly specified by their links to the index files, as specified when the index file was opened (OP.xbLink). INSERT_XB can process up to 256 index files per call. Since each index file requires a data file, this means that up to 256 data files can be processed per call, as well. Also possible is that all 256 index handles refer to the same, single data file. Yet another possibility is that there is 1 index file, and so 1 data file. The possibilities can include those and anything in between.

Example: Specifying a single index file

The simplest form is where a single index handle is specified. This implies a single data file, too. AccessPack setup for this is:

```
AP.func = INSERT_XB;
AP.handle = indexHandle;
AP.recNo = 0;
AP.recPtr = &recordStruct; // contents referred to below as *recordStruct
AP.keyPtr = keyBuffer;
AP.nextPtr = NULL;
```

A call to BULLET with the above does the following:

1. The data in *recordStruct is used as a new record that is appended to the data file. The data file was linked to this index during the index open, in OP.xbLink.
2. A key is built by BULLET, based on the data in *recordStruct, and that key is inserted into the index file (AP.handle). Stored with the key is the record number of the record added above.

Note: AP.recNo must be set to 0 prior to the call. Any positive number results in an error (0x80000000, and negative numbers, may be used when more than one AP pack is used - see below).

Upon return, if no error, the return code is 0. AP.recNo is set to the physical record number in the data file that *recordStruct was placed. The key that was stored, including any enumerator, is in *keyBuffer.

Upon return, and there was an error, the return code is either -1 or 1. If -1, the error was caused during processing of the data file portion, and the error code itself is in AP.stat. If +1, the error was caused during processing of the index file, and the error code itself is in AP.stat, as well. The return code is, as in all BULLET transaction-list routines, an index of the AP pack that generated the error — negative if a data file error, positive if an index file error. Since this example has only the single pack, only a -1 or +1 could be returned, or 0.

Note: If an error occurred after any part of the database had changed (during this particular call), then any and all changes that were made are backed-out, and the files restored to the same state as before the call.

Example: Specifying two index files for a single data file

Two index files, related to the same data file, would set AccessPack to:

```
AP[0].func = INSERT_XB;
AP[0].handle = indexHandle_0;
AP[0].recNo = 0;
AP[0].recPtr = &recordStruct;
AP[0].keyPtr = keyBuffer_0;
AP[0].nextPtr = AP[1];

AP[1].handle = indexHandle_1;
AP[1].recNo = 0x80000000;
AP[1].recPtr = &recordStruct;
AP[1].keyPtr = keyBuffer_1;
AP[1].nextPtr = NULL;
```

A call to BULLET with the above does the following:

1. The data in *recordStruct is used as a new record that is appended (added) to the data file.
2. A key is built by BULLET, based on the data in *recordStruct, and that key is inserted into the index file (AP[0].handle). Stored with the key is the record number of the record added above.
3. A second key is built by BULLET, based on the data in *recordStruct, and that key is inserted into the second index file (AP[1].handle). Stored with the key is the record number of the record added above.

Note: The 0x80000000 in AP[1].recNo signifies that AP[1] is using the same data record that was appended during processing of AP[0]. This results in just the one data record being added. AP[1].recPtr must still, however, point to the same data as AP[0].recPtr does.

Upon return, if no error, the return code is 0. AP[0].recNo is set to the physical record number in the data file that *recordStruct was placed. The key that was stored for the first index, including any enumerator, is in the buffer at AP[0].keyPtr. AP[1].recNo is set to the same physical record number as AP[0].recNo, **except** that the record number is negative: For example, if AP[0].recNo is 22 on return, AP[1].recNo is -22 (the original 0x80000000 value is overwritten). The key that was stored for the second index, including any enumerator, is in the buffer at AP[1].keyPtr.

Upon return, and there was an error, the return code can be -2, -1, 1, or 2. If negative, the error was caused during processing of that AP pack's data file portion, and the error code itself is in AP[abs(rez)-1].stat (where rez is the return code, and -1 since C arrays start at 0). If the return code was positive, the error was caused during processing of that AP pack's index file, and the error code itself is in AP[rez-1].stat, as well. The return code is, as in all BULLET transaction-list routines, an index of the AP pack that generated the error — negative if a data file error, positive if an index file error.

Note: If an error occurred after any part of the database had changed (during this particular call), then any and all changes that were made are backed-out, and the files restored to the same state as before the call.

Example: Specifying two index files for each of two different data files

Four total files: two index files related to one data file, and two other index files related to another data file, would set AccessPack to:

```
AP[0].func = INSERT_XB;
AP[0].handle = indexHandle_0;
AP[0].recNo = 0;
AP[0].recPtr = &recordStruct_0;
AP[0].keyPtr = keyBuffer_0;
AP[0].nextPtr = AP[1];

AP[1].handle = indexHandle_1;
AP[1].recNo = 0x80000000;
AP[1].recPtr = &recordStruct_0;
AP[1].keyPtr = keyBuffer_1;
AP[1].nextPtr = AP[2];

AP[2].handle = indexHandle_2;
AP[2].recNo = 0;
AP[2].recPtr = &recordStruct_1;
AP[2].keyPtr = keyBuffer_2;
AP[2].nextPtr = AP[3];

AP[3].handle = indexHandle_3;
AP[3].recNo = 0x80000000;
AP[3].recPtr = &recordStruct_1;
AP[3].keyPtr = keyBuffer_3;
AP[3].nextPtr = NULL;
```

A call to BULLET with the above does the following:

1. The data in *recordStruct_0 is used as a new record that is appended to the data file linked to the index file in AP[0].handle.
2. A key is built by BULLET, based on the data in *recordStruct_0, and that key is inserted into the index file (AP[0].handle). Stored with the key is the record number of the record added above, for _0.
3. A second key is built by BULLET, based on the data in *recordStruct_0, and that key is inserted into the second index file (AP[1].handle). Stored with the key is the record number of the record added above, using *recordStruct_0.
4. The data in *recordStruct_1 is used as a new record that is appended to the data file linked to the index file in AP[2].handle.
5. A third key is built by BULLET, based on the data in *recordStruct_1, and that key is inserted into the index file (AP[2].handle). Stored with the key is the record number of the record added above, for _1.
6. A fourth key is built by BULLET, based on the data in *recordStruct_1, and that key is inserted into the fourth index file (AP[3].handle). Stored with the key is the record number of the record added above, using *recordStruct_1.

Note: The 0x80000000 in AP[1].recNo signifies that AP[1] is using the same data record that was appended during processing of AP[0]. This results in just the one data record being added. AP[1].recPtr must still, however, point to the same data as AP[0].recPtr does. The same applies to AP[2] and AP[3] (though different values, of course).

Upon return, if no error, the return code is 0. AP[0].recNo is set to the physical record number in the data file that *recordStruct_0 was placed. The key that was stored for the first index, including any enumerator, is in the buffer at AP[0].keyPtr. AP[1].recNo is set to the same physical record number as AP[0].recNo, **except** that the record number is negative: For example, if AP[0].recNo is 22 on return, AP[1].recNo is -22 (the original 0x80000000 value is overwritten). The key that was stored for the second index, including any enumerator, is in the buffer at AP[1].keyPtr. AP[2].recNo is set to the physical record number in the data file that *recordStruct_1 was placed. The key that was stored for the third index, including any enumerator, is in the buffer at AP[2].keyPtr. AP[3].recNo is set to the same physical record number as AP[2].recNo, **except** that the record number is negative: For example, if
AP[2].recNo is 74 on return, AP[3].recNo is -74 (the original 0x80000000 value is overwritten). The key that was stored for the fourth index, including any enumerator, is in the buffer at AP[3].keyPtr.

Upon return, and there was an error, the return code can be -4 to -1, or 1 to 4. If negative, the error was caused during processing of that AP pack's data file portion, and the error code itself is in AP[abs(rez)-1].stat (where rez is the return code, and -1 since C arrays start at 0). If the return code was positive, the error was caused during processing of that AP pack's index file, and the error code itself is in AP[rez-1].stat, as well. The return code is, as in all BULLET transaction-list routines, an index of the AP pack that generated the error — negative if a data file error, positive if an index file error.

Note: If an error occurred after any part of the database had changed (during this particular call), then any and all changes that were made are backed-out, and the files restored to the same state as before the call.

Example: Specifying two index files for two records in the same data file

Three files: two index files related to one data file, where two data records are to be appended, would set AccessPack to:

```
AP[0].func = INSERT_XB;
AP[0].handle = indexHandle_0;
AP[0].recNo = 0;
AP[0].recPtr = &recordStruct_0;
AP[0].keyPtr = keyBuffer_0;
AP[0].nextPtr = AP[1];

AP[1].handle = indexHandle_1;
AP[1].recNo = 0x80000000;
AP[1].recPtr = &recordStruct_0;
AP[1].keyPtr = keyBuffer_1;
AP[1].nextPtr = AP[2];

AP[2].handle = indexHandle_0;
AP[2].recNo = 0;
AP[2].recPtr = &recordStruct_1;
AP[2].keyPtr = keyBuffer_2;
AP[2].nextPtr = AP[3];

AP[3].handle = indexHandle_1;
AP[3].recNo = 0x80000000;
AP[3].recPtr = &recordStruct_1;
AP[3].keyPtr = keyBuffer_3;
AP[3].nextPtr = NULL;
```

A call to BULLET with the above does the following:

1. The data in *recordStruct_0 is used as a new record that is appended to the data file linked to the index file in AP[0].handle.
2. A key is built by BULLET, based on the data in *recordStruct_0, and that key is inserted into the index file (AP[0].handle). Stored with the key is the record number of the record added above, for _0.
3. A second key is built by BULLET, based on the data in *recordStruct_0, and that key is inserted into the second index file (AP[1].handle). Stored with the key is the record number of the record added above, using *recordStruct_0.
4. The data in *recordStruct_1 is used as a new record that is appended to the data file linked to the index file in AP[2].handle. Since AP[2].handle is the same index file as that of AP[0].handle, this means it's also the same data file as was just operated on above — a second data record is appended to the data file. The net effect of this operation is to call INSERT_XB twice, once for one insert, then again for the second. The difference is that the operation is atomic — if one fails, the other is not committed; it's an "all or nothing" operation.
5. A third key is built by BULLET, based on the data in *recordStruct_1, and that key is inserted into the index file (AP[2].handle). Stored with the key is the record number of the record added directly above, for _1. Note that this index file is the same as specified in AP[0].handle.
6. A fourth key is built by BULLET, based on the data in *recordStruct_1, and that key is inserted into the fourth index file (AP[3].handle). Stored with the key is the record number of the record added above, using *recordStruct_1.

The return ritual is as described above, for "*Specifying two index files each for two different data files*".

Example: Specifying a single index file for a previously added data record

This form lets you insert a key without adding a data record. This would be required if you were, for example, creating a temporary index of select records in a data file (i.e., the data records already exist, you just want to index them). AccessPack setup for this is:

```
AP.func = INSERT_XB;
AP.handle = indexHandle;
AP.recNo = -recordNumberOfExistingRecord;
AP.recPtr = &recordStruct;
AP.keyPtr = keyBuffer;
AP.nextPtr = NULL;
```

A call to BULLET with the above does the following:

A key is built by BULLET, based on the data in *recordStruct, and that key is inserted into the index file (AP.handle). Stored with the key is the absolute value of the record number specified in AP.recNo (which is set to negative record number).

Upon return, if no error, the return code is 0. AP.recNo is changed to abs(AP.recNo). The key that was stored, including any enumerator, is in *keyBuffer. No data file access is made.

Upon return, and there was an error, the return code is either -1 or 1. If -1, the error was caused during processing of the data file portion, and the error code itself is in AP.stat. If +1, the error was caused during processing of the index file, and the error code itself is in AP.stat, as well. The return code is, as in all BULLET transaction-list routines, an index of the AP pack that generated the error — negative if a data file error, positive if an index file error. Since this example has only the single pack, only a -1 or +1 could be returned.

Note: If an error occurred after any part of the database had changed (during this particular call), then any and all changes that were made are backed-out, and the files restored to the same state as before the call.

An example use of this INSERT_XB feature is to create an *ad hoc* index of, say, records marked as deleted. To do this, create a new index file (say, with a key of NAME). Get each data record, by record number using GET_RECORD_XB (for records 1 to number-of-records), and check the .tag byte. If '*', call INSERT_XB with the *negative* value of AP.recNo. Do this for every such marked record. After all records are processed, you have an index of all deleted records in the data file. Delete the index when no longer needed. That's just one example.

UPDATE_XB

Uses ACCESSPACK

```
   IN                       OUT
AP.func                  AP.stat
AP.handle                *AP.keyPtr
AP.recNo
AP.recPtr
AP.keyPtr
AP.nextPtr
```

Update any and all files in the transaction list if necessary, including both index and data files.

This routine is used to update data records while also updating the index files if a key field has changed due to data record updates. Up to 256 index files may be updated per call, as well as 256 data files, too, for a total of 512 files managed per single UPDATE_XB call.

Only index handles are listed in AP.handle. Each index file has associated with it a data file, known internally to BULLET (the xbLink from OPEN_XB). There may be more than one index file for a data file, but there is always one data file per index handle specified in the list. In other words, you can list five index files, each indexing the same xbLink data file, and have BULLET perform an **atomic** update of that list. Or, another possibility is that you have a single index file, indexing a single data file. Or, you can list 256 index files, each indexing a single data file (512 total files).

This and several other routines are transaction-list-based. This means that 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 index files) effectively restored to its original state.

If the routine failed to complete, the function return value is the number (1-based) of the pack that caused the failure. A positive number indicates the failure was from an index operation; a negative number indicates the failure was from a data operation. In each case, the absolute value of the return code is the list item that failed (the pack index). For example, if five index handles are in the list(AP[0] to AP[4]), and an error occurred on the last pack's index file, the return code would be positive 5, indicating the fifth pack (AP[4]) failed. Since it was a positive 5, the index file was being processed when the error occurred. *Being processed* means not only physical access, but verification, etc. If the return code was -5, then again, the error was in the fifth pack, but since it is negative, the error occurred while processing the data file. In either case, upon return, the database is restored to the way it was before the UPDATE_XB call was made. Remedy the error, if possible, and UPDATE_XB again.

Each pack **must** include a separate key buffer. You must not share a common key buffer. Doing so disables any chance of recovering the index files in case of error, since it is in these buffers that BULLET places any newly built keys, and it is from these that BULLET, upon an error condition, deletes these keys (required for roll-back).

The enumerator is automatically maintained by this routine, if required (DUPS_ALLOWED and the key already exists with enumerator 0). The process is the same as INSERT_XB's.

How an update works

All data records specified in the list are read from disk into memory, except those with AP.recNo=0. Therefore, a **memory allocation** large enough to store all unique data records is made upon entry to this routine (and released at exit). For example, if the list includes two implicit data files, and the record lengths of those two data files are 2048 and 4096 bytes, an allocation of 6K is made. In addition, 40KB more is allocated for workspace. So, for this example, 46K is allocated (rounded up to 48KB, the next 4KB page boundary). Since up to 256 unique records are possible, where a unique record is identified by handle/record number, be aware of the memory requirements if you are updating very large databases (e.g., 256 unique records, each 4KB in length, would have UPDATE_XB allocate a bit over 1MB of memory for this call).

After the data records have been read from disk, each list-item is processed, in order. The disk record image previously read is compared with the record image at AP.recPtr. If the same, that item is skipped, and the next item in the list is processed. If you know beforehand that that record is the same, set that item's AP.recNo=0 so you can avoid having its disk image read and stored (or do not include it in the list at all). If the images differ, BULLET creates a key for the index file being processed, for each record image (the original and the one in AP.recPtr). If the keys generated are the same, no index file update is needed. If different, the original key for that record is deleted from that index file, and the new key inserted. Finally, the new record replaces the old, the new directly overwriting the original. Note that the actual sequence of the update event differs somewhat from this description in order to optimize the process.

Specifying Files

As mentioned, only the index file handles are specified in AP.handle. Data files are implicitly specified by their links to the index files, as specified when the index file was opened (OP.xbLink). UPDATE_XB can process up to 256 index files per call. Since each index file requires a data file, this means that up to 256 data files can be processed per call as well. Also possible is that all 256 index handles refer to the same, single data file. Yet another possibility is that there is 1 index file, and so 1 data file. The possibilities can include those and anything in between.

Example: Specifying a single index file

The simplest form is where a single index handle is specified. This implies a single data file, too. AccessPack setup for this is:

```
AP.func = UPDATE_XB;
AP.handle = indexHandle;
AP.recNo = recordToUpdate;
AP.recPtr = &recordStruct;
AP.keyPtr = keyBuffer;
AP.nextPtr = NULL;
```

A call to BULLET with the above does the following:

1. The data in *recordStruct is used as the new record that is to replace the data record at AP.recNo. The data file was linked to this index file (AP.handle) during the index open, in OP.xbLink.
2. If the record data in *recordStruct is the same as the original disk record, nothing is done. If the data is new, the key fields are compared to that belonging to the original disk record, and if the same, only the record data is updated. If the new record's key differs from the original's, the original key for this record is removed from the index, and the new key inserted.

AP.recNo must be set to the record number that you are updating. Any GET_XB routine (GET_EQUAL_XB, etc.) may be used to identify the number of a data record. Key access has the obvious advantage of knowing the record number of a specific key (for example, Betty Barbar's data). Any record number, from 1 to number of records in the data file, can be used. In addition, a negative record number can be used. This is treated exactly the same as a positive record number (the absolute value is used). The reason this is allowed is because INSERT_XB replaces 0x80000000 record numbers with the negative value of the previous insert.

Upon return, if no error, the return code is 0. If the record data was new, the key for that data record, including any enumerator, is in *keyBuffer. This is so even if key fields had not changed.

Upon return, and there was an error, the return code is either -1 or 1. If -1, the error was caused during processing of the data file portion, and the error code itself is in AP.stat. If +1, the error was caused during processing of the index file, and the error code itself is in AP.stat, as well. The return code is, as in all BULLET transaction-list routines, an index of the AP pack that generated the error — negative if a data file error, positive if an index file error. Since this example has only the single pack, only a -1 or +1 could be returned.

Note: If an error occurred after any part of the database had changed (during this particular call), then any and all changes that were made are backed-out, and the files restored to the same state as before the call.

Example: Specifying two index files for a single data file

Two index files, related to the same data file, would set AccessPack to:

```
AP[0].func = UPDATE_XB;
AP[0].handle = indexHandle_0;
AP[0].recNo = recordToUpdate;
AP[0].recPtr = &recordStruct;
AP[0].keyPtr = keyBuffer_0;
AP[0].nextPtr = AP[1];

AP[1].handle = indexHandle_1;
AP[1].recNo = recordToUpdate;
AP[1].recPtr = &recordStruct;
AP[1].keyPtr = keyBuffer_1;
AP[1].nextPtr = NULL;
```

A call to BULLET with the above does the following:

1. The data in *recordStruct is used as the new record that is to replace the data record at AP.recNo. The data file was linked to this index file (AP.handle) during the index open, in OP.xbLink.
2. If the record data in *recordStruct is the same as the original disk record, nothing is done. If the data is new, the key fields are compared to that belonging to the original disk record, and if the same, only the record data is updated. If the new record's key differs from the original's, the original key for this record is removed from the index, and the new key inserted.
3. The operation performed directly above is repeated, this time for the second index file. The new record data, and the record number to update are, for this particular example, the same.

AP.recNo must be set to the record number that you are updating. Each AP[].recNo must be set to a valid record number, even if the record number is the same as the previous AP[] pack's (the case where you have more than one index file for a data file). BULLET knows if the record number duplicates a number in a previous AP pack, and allocates resources for only the first encounter of the data record. Subsequent encounters refer to the first.

Upon return, if no error, the return code is 0. If the new and original data records differ, the key for the new data record, including any enumerator, is in the buffer at AP[0].keyPtr. This even if the key fields did not change. The same applies to the second index, with the new data key in AP[1].keyPtr.

Upon return, and there was an error, the return code can be -2, -1, 1, or 2. If negative, the error was caused during processing of that AP pack's data file portion, and the error code itself is in AP[abs(rez)-1].stat (where rez is the return code, and -1 since C arrays start at 0). If the return code was positive, the error was caused during processing of that AP pack's index file, and the error code itself is in AP[rez-1].stat, as well. The return code is, as in all BULLET transaction-list routines, an index of the AP pack that generated the error — negative if a data file error, positive if an index file error.

Note: If an error occurred after any part of the database had changed (during this particular call), then any and all changes that were made are backed-out, and the files restored to the same state as before the call.

Example: Specifying two index files for each of two different data files

Four total files: two index files related to one data file, and two other index files related to another data file, would set AccessPack to:

```
AP[0].func = UPDATE_XB;
AP[0].handle = indexHandle_0;
AP[0].recNo = recordToUpdate_0;
AP[0].recPtr = &recordStruct_0;
AP[0].keyPtr = keyBuffer_0;
AP[0].nextPtr = AP[1];

AP[1].handle = indexHandle_1;
AP[1].recNo = recordToUpdate_0;
AP[1].recPtr = &recordStruct_0;
AP[1].keyPtr = keyBuffer_1;
AP[1].nextPtr = AP[2];

AP[2].handle = indexHandle_2;
AP[2].recNo = recordToUpdate_1;
AP[2].recPtr = &recordStruct_1;
AP[2].keyPtr = keyBuffer_2;
AP[2].nextPtr = AP[3];

AP[3].handle = indexHandle_3;
AP[3].recNo = recordToUpdate_1;
AP[3].recPtr = &recordStruct_1;
AP[3].keyPtr = keyBuffer_3;
AP[3].nextPtr = NULL;
```

A call to BULLET with the above does the following:

1. The data in *recordStruct_0 is used as the new record that is to replace the data record at AP[0].recNo in the data file linked to the index file in AP[0].handle.
2. If the record data in *recordStruct_0 is the same as the original disk record, nothing is done. If the data is new, the key fields are compared to that belonging to the original disk record, and if the same, only the record data is updated. If the new record's key differs from the original's, the original key for this record is removed from the index, and the new key inserted.
3. The operation performed directly above is repeated, this time for the second index file. The new record data, and the record number to update, are for this particular example, the same.
4. The data in *recordStruct_1 is used as the new record that is to replace the data record at AP[2].recNo in the data file linked to the index file in AP[2].handle.
5. If the record data in *recordStruct_1 is the same as the original disk record, nothing is done. If the data is new, the key fields are compared to that belonging to the original disk record, and if the same, only the record data is updated. If the new record's key differs from the original's, the original key for this record is removed from the index, and the new key inserted.
6. The operation performed directly above is repeated, this time for the fourth index file. The new record data, and the record number to update are, for this particular example, the same.

Upon return, if no error, the return code is 0. If the new and original data records differ, the keys for the new data records, including any enumerators, are in the buffers at AP[0].keyPtr to AP[3].keyPtr. This even if the key fields did not change. If one, or all, of the new data records matched the original data record, nothing is placed in *keyBuffer for that index.

Upon return, and there was an error, the return code can be -4 to -1, or 1 to 4. If negative, the error was caused during processing of that AP pack's data file portion, and the error code itself is in AP[abs(rez)-1].stat (where rez is the return code, and rez-1 since C arrays start at 0). If the return code was positive, the error was caused during processing of that AP pack's index file, and the error code itself is in AP[rez-1].stat, as well. The return code is, as in all BULLET transaction-list routines, an index of the AP pack that generated the error — negative if a data file error, positive if an index file error.

Note: If an error occurred after any part of the database had changed (during this particular call), then any and all changes that were made are backed-out, and the files restored to the same state as before the call.

Example: Specifying two index files for two records in the same data file

Three files: two index files related to one data file, where two data records are to be updated, would set AccessPack to:

```
AP[0].func = UPDATE_XB;
AP[0].handle = indexHandle_0;
AP[0].recNo = recordToUpdate_0;
AP[0].recPtr = &recordStruct_0;
AP[0].keyPtr = keyBuffer_0;
AP[0].nextPtr = AP[1];

AP[1].handle = indexHandle_1;
AP[1].recNo = recordToUpdate_0;
AP[1].recPtr = &recordStruct_0;
AP[1].keyPtr = keyBuffer_1;
AP[1].nextPtr = AP[2];

AP[2].handle = indexHandle_0;
AP[2].recNo = recordToUpdate_1;
AP[2].recPtr = &recordStruct_1;
AP[2].keyPtr = keyBuffer_2;
AP[2].nextPtr = AP[3];

AP[3].handle = indexHandle_1;
AP[3].recNo = recordToUpdate_1;
AP[3].recPtr = &recordStruct_1;
AP[3].keyPtr = keyBuffer_3;
AP[3].nextPtr = NULL;
```

A call to BULLET with the above does the following:

1. The data in *recordStruct_0 is used as the new record that is to replace the data record at AP[0].recNo in the data file linked to the index file in AP[0].handle.
2. If the record data in *recordStruct_0 is the same as the original disk record, nothing is done. If the data is new, the key fields are compared to that belonging to the original disk record, and if the same, only the record data is updated. If the new record's key differs from the original's, the original key for this record is removed from the index, and the new key inserted.
3. The operation performed directly above is repeated, this time for the second index file. The new record data, and the record number to update, are for this particular example, the same.

4. The data in *recordStruct_1 is used as the new record that is to replace the data record at AP[2].recNo in the data file linked to the index file in AP[2].handle. This is the same index file as the first AP pack, and also the same data file. However, this is a different record number.
5. If the record data in *recordStruct_1 is the same as the original disk record, nothing is done. If the data is new, the key fields are compared to that belonging to the original disk record, and if the same, only the record data is updated. If the new record's key differs from the original's, the original key for this record is removed from the index, and the new key inserted.
6. The operation performed directly above is repeated, this time for the fourth index file. The new record data, and the record number to update are, for this particular example, the same.

The return ritual is as described above, for "*Specifying two index files each for two different data files*".

REINDEX_XB

Uses ACCESSPACK

IN	OUT
`AP.func`	`AP.stat`
`AP.handle`	`AP.recNo`
`AP.keyPtr`	`*AP.keyPtr`
`AP.nextPtr`	

Reindex all files in the transaction list, re-evaluating the key expression in the process.

This routine is used to reindex up to 256 index files per call. The index files must already exist and be open. Any existing key values are overwritten by new key data. In other words, if you have a 100MB index file, REINDEX_XB uses the same file space, building new keys over 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. Another use for creating a new index file and reindexing to that is to, after creating it (COPY_INDEX_HEADER_XB can be used), use EXPAND_FILE_DOS and expand it to the expected size. This has the benefit of ensuring that this file allocation is as contiguous as the file system allows (without relying on OS/API-specific calls).

If the routine failed to complete, the function return value is the number (1-based) of the pack that caused the failure. A positive number indicates the failure was from an index operation; a negative number indicates the failure was from a data operation (reading the data file). In each case, the absolute value of the return code is the list item that failed (the pack index). For example, if five index handles are in the list(AP[0] to AP[4]), and an error occurred on the last pack's index file, the return code would be positive 5, indicating the fifth pack (AP[4]) failed. Since it was a positive 5, the index file was being processed when the error occurred. *Being processed* means not only physical access, but verification, etc. If the return code was -5, then again, the error was in the fifth pack, but since it is negative, the error occurred while processing the data file.

Unlike INSERT_XB & UPDATE_XB, each pack need not include a separate key buffer; you may share a common key buffer. If duplicate keys are generated in the reindex process and the sort function does not flag DUPS_ALLOWED, an error is returned. The duplicate key is in *AP.keyPtr and the record number it was generated from in AP.recNo (both these are valid only on error). Since no roll-back is performed, there is only a real need for a single key buffer. You may use separate ones, too.

This routine creates a **temporary** work file in either the current directory or, if the environment variable TMP is defined, in the directory pointed to by TMP=. The path used for this temporary file may also be specified at run-time by using the TMP_PATH_PTR item for SET_SYSVARS_XB. If TMP_PATH_PTR is NULL (default), then TMP=

is used, or if that is not found, then the current directory is used. The size of this temporary file is, in bytes, approximately (keylength+4) * number of records in the data file. The resultant index files are, by default, optimized for minimum size *and* maximum retrieval speed. This full-node packing leaves one empty key per node, which means b-tree splitting will occur almost immediately upon inserting data (with INSERT_XB, or STORE_KEY_XB).

This behaviour can be modified with the REINDEX_PACK_PCT item for SET_SYSVARS_XB so that less packing is done. Less packing would improve subsequent INSERT_XB performance since all nodes are not almost full as they are with a full pack. File size and retrieval times increase, though, but perhaps not noticeably.

During the reindex process, each record is checked for a matching skip-tag value, as set in SET_SYSVARS_XB. The skip-tag is set to 0 by default, where no check is done and keys for all records in the data file are inserted into the index file.

Network Level

- LOCK_XB
- UNLOCK_XB
- LOCK_INDEX_XB
- UNLOCK_INDEX_XB
- LOCK_DATA_XB
- UNLOCK_DATA_XB
- CHECK_REMOTE_XB
- RELOCK_XB
- RELOCK_INDEX_XB
- RELOCK_DATA_XB

LOCK_XB

Uses LOCKPACK

IN	OUT
LP.func	LP.stat
LP.handle	
LP.xlMode	
LP.dlMode	
LP.recStart=0	
LP.nextPtr	

Lock all bytes of the files in the list for exclusive use by the current process, and reload file headers from disk. LP.recStart must be 0 for each pack.

This routine is used to lock the database for either exclusive use by this process, or shared access (allowing any process to read, but not write, to the files). Up to 256 index files may be locked per call, as well as 256 data files, too, for a total of 512 files per single LOCK_XB call. Shared-access locking prevents all processes from writing to the file while a shared lock is in force, including this process. To relock in exclusive lock mode, without unlocking first, use: RELOCK_XB.

Only index handles are listed in LP.handle. Each index file has associated with it a data file, known internally to BULLET (the xbLink from OPEN_XB). There may be more than one index file for a data file, but there is always one data file per index handle specified in the list. For example, you can list five index files, each indexing the same xbLink data file, and have BULLET perform an **atomic** lock of that list.

LP.xlMode is set to 1 to perform a shared lock on the index file. Set to 0 for an exclusive lock. A shared lock allows only reading.

LP.dlMode is set to 1 to perform a shared lock on the data file. Set to 0 for an exclusive lock. A shared lock allows only reading.

The lock mode (shared <-> exclusive) can be changed using RELOCK_XB.

Bullet maintains a per-handle lock count, and does a physical region lock only upon the first lock request (or on a relock request). It is only on this first lock request that the header is reloaded. When the lock count returns to 0 (from UNLOCK_XB calls), it is at that time the header is flushed, if required. Only full-locks are maintained in this way. The number of outstanding locks can be determined from the SIP and SDP structures, from the STAT_XB routines. Note that individual LOCK_INDEX_XB and UNLOCK_INDEX_XB routines, as well as the data ones, also act upon this lock count. Therefore, you can lock a file 100 times in a row, but only on the first lock are any operations actually performed, and only on the last unlock are any performed. Other lock/unlock calls (other than the first lock or last unlock) simply increment or decrement the lock count for that handle.

This and several other routines are transaction-list-based. This means that if a failure occurs prior to the routine's completion, all locks made to the database by this routine will be unlocked.

If the routine failed to complete, the function return value is the number (1-based) of the pack that caused the failure. A positive number indicates the failure was from an index operation; a negative number indicates the failure was from a data operation. In each case, the absolute value of the return code is the list item that failed (the pack index). For example, if five index handles are in the list(AP[0] to AP[4]), and an error occurred on the last pack's index file, the return code would be positive 5, indicating the fifth pack (AP[4]) failed. Since it was a positive 5, the index file was being processed when the error occurred. *Being processed* means not only physical access, but verification, etc. If the return code was -5, then again, the error was in the fifth pack, but since it is negative, the error occurred while processing the data file. In either case, upon return, any files locked during this call are unlocked before returning.

The advantage of using region locks (LOCK_XB locks entire file regions) 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.

Using the operating system to control access also prevents other processes from accessing the files. Other methods, such as internal locking (such as using 'L' in the tag field as a program-aware in-use flag), work fine so long as each process accessing the files knows about this internal "locking". However, since it's proprietary, other processes may not know about it. Any locking system that is not commonly shared throughout the system is not effective when it comes to preventing other processes from corrupting files.

Note: Region locking prevents other processes from both writing and reading the locked file. For operating systems that do not provide shared locks, and read-access is required at all times, you may specify this type access with the access-sharing mode when the BULLET file is opened. Once opened for this (R/W, DenyWrite) then only the current process can write to the file until it is closed. Other processes must open the file for Read-Only access. For small networks (two or three nodes), this may be suitable. Otherwise, region locking is much preferred, and very much faster, since files do not have to be opened and closed every time the access state needs to change.

UNLOCK_XB

Uses LOCKPACK

IN	OUT
`LP.func`	`LP.stat`
`LP.handle`	
`LP.recStart=0`	
`LP.nextPtr`	

Unlock all bytes in the specified files (previously locked) and flush the files' headers to disk (the flush is done before the locks are released). Also unlock all bytes in the related data file and flush the data file header to disk. LP.recStart must be 0 for each pack.

Note: If a shared-lock is active for this handle (as set by this process), the flush is not performed. This because writing to the locked region is not permitted (nor is the flush required since nothing could have been changed).

If the routine failed to complete, the function return value is the number (1-based) of the pack that caused the failure. A positive number indicates the failure was from an index operation; a negative number indicates the failure was from a data operation. In each case, the absolute value of the return code is the list item that failed (the pack index). For example, if five index handles are in the list(AP[0] to AP[4]), and an error occurred on the last pack's index file, the return code would be positive 5, indicating the fifth pack (AP[4]) failed. Since it was a positive 5, the index file was being processed when the error occurred. *Being processed* means not only physical access, but verification, etc. If the return code was -5, then again, the error was in the fifth pack, but since it is negative, the error occurred while processing the data file.

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), the remaining files must be manually unlocked with the UNLOCK_KEY_XB and UNLOCK_DATA_XB routines.

Bullet maintains a per-handle lock count, and does a physical region lock only upon the first lock request (or on a relock request). It is only on this first lock request that the header is reloaded. When the lock count returns to 0 (from UNLOCK_XB calls), it is at that time the header is flushed, if required. Only full-locks are maintained in this way. The number of outstanding locks can be determined from the SIP and SDP structures, from the STAT_XB routines. Note that individual LOCK_INDEX_XB and UNLOCK_INDEX_XB routines, as well as the data ones, also act upon this lock count. Therefore, you can lock a file 100 times in a row, but only on the first lock are any operations actually performed, and only on the last unlock are any performed. Other lock/unlock calls (other than the first lock or last unlock) simply increment or decrement the lock count for that handle.

LOCK_INDEX_XB

Uses LOCKPACK

IN	OUT
`LP.func`	`LP.stat`
`LP.handle`	
`LP.xlMode`	

Lock all bytes of the index file for exclusive use by the current process and reload the index file's header from disk.

LP.xlMode is set to 1 to perform a shared lock. Set to 0 for an exclusive lock. A shared lock allows only reading. The lock mode (shared <-> exclusive) can be changed using RELOCK_INDEX_XB.

Bullet maintains a per-handle lock count, and does a physical region lock only upon the first lock request (or on a relock request). It is only on this first lock request that the header is reloaded. When the lock count returns to 0 (from UNLOCK_XB calls), it is at that time the header is flushed, if required. Only full-locks are maintained in this way. The number of outstanding locks can be determined from the SIP and SDP structures, from the STAT_XB routines. Note that individual LOCK_INDEX_XB and UNLOCK_INDEX_XB routines, as well as the data ones, also act upon this lock count. Therefore, you can lock a file 100 times in a row, but only on the first lock are any operations actually performed, and only on the last unlock are any performed. Other lock/unlock calls (other than the first lock or last unlock) simply increment or decrement the lock count for that handle.

UNLOCK_INDEX_XB

Uses LOCKPACK

IN	OUT
`LP.func`	`LP.stat`
`LP.handle`	

Unlock all bytes in the specified file (previously locked) and flush the file's header to disk (the flush is done before the locks are released).

If the current lock state is shared, the flush is not performed.

Bullet maintains a per-handle lock count, and does a physical region lock only upon the first lock request (or on a relock request). It is only on this first lock request that the header is reloaded. When the lock count returns to 0 (from UNLOCK_XB calls), it is at that time the header is flushed, if required. Only full-locks are maintained in this way. The number of outstanding locks can be determined from the SIP and SDP structures, from the STAT_XB routines. Note that individual LOCK_INDEX_XB and UNLOCK_INDEX_XB routines, as well as the data ones, also act upon this lock count. Therefore, you can lock a file 100 times in a row, but only on the first lock are any operations actually performed, and only on the last unlock are any performed. Other lock/unlock calls (other than the first lock or last unlock) simply increment or decrement the lock count for that handle.

LOCK_DATA_XB

Uses LOCKPACK

IN	OUT
`LP.func`	`LP.stat`
`LP.handle`	
`LP.dlMode`	
`LP.recStart`	
`LP.recCount`	

Lock all bytes of the data file specified in LP.handle for exclusive use by the current process. It also reloads the data file header from disk. You must set LP.recStart=0 to lock all bytes. To lock a single record, or a set of contiguous records, set LP.recStart to the first record to lock and LP.recCount to the number of records to lock.

Header re-loading is performed only if locking all bytes.

If LP.recStart is not 0, be aware that the header is not locked, nor is it re-loaded. Also, maintaining a lock on the single record prevents any other process from doing a full lock on that data file, thereby preventing write access to the file from any other BULLET process using LOCK_XB, but *not necessarily preventing other applications* from writing to that file. That may or may not be good. It does not prevent other BULLET processes from reading that file (except for that locked record).

Multiple single records are allowed, but each must then be unlocked individually, in the same format (start, count) as the lock.

LP.dlMode is set to 1 to perform a shared lock. Set to 0 for an exclusive lock. A shared lock allows only reading. The lock mode (shared <-> exclusive) can be changed using RELOCK_DATA_XB.

Bullet maintains a per-handle lock count, and does a physical region lock only upon the first lock request (or on a relock request). It is only on this first lock request that the header is reloaded. When the lock count returns to 0 (from UNLOCK_XB calls), it is at that time the header is flushed, if required. Only full-locks are maintained in this way. The number of outstanding locks can be determined from the SIP and SDP structures, from the STAT_XB routines. Note that individual LOCK_INDEX_XB and UNLOCK_INDEX_XB routines, as well as the data ones, also act upon this lock count. Therefore, you can lock a file 100 times in a row, but only on the first lock are any operations actually performed, and only on the last unlock are any performed. Other lock/unlock calls (other than the first lock or last unlock) simply increment or decrement the lock count for that handle.

UNLOCK_DATA_XB

Uses LOCKPACK

```
   IN                        OUT
LP.func                   LP.stat
LP.handle
LP.recStart
LP.recCount
```

Unlock all bytes in the specified file handle (previously locked) and flush the data file header to disk (the flush is done before the lock is released). You must set LP.recStart=0 to unlock all bytes. To unlock a single record, or a set of contiguous records, set LP.recStart to the first record to unlock and LP.recCount to the number of records to unlock.

Header flushing is performed only if unlocking a full lock.

An unlock must exactly mimic its corresponding lock. This means if you lock several records singly, you must unlock each of those records — you cannot use LP.recStart=0 to unlock singly-locked records.

Bullet maintains a per-handle lock count, and does a physical region lock only upon the first lock request (or on a relock request). It is only on this first lock request that the header is reloaded. When the lock count returns to 0 (from UNLOCK_XB calls), it is at that time the header is flushed, if required. Only full-locks are maintained in this way. The number of outstanding locks can be determined from the SIP and SDP structures, from the STAT_XB routines. Note that individual LOCK_INDEX_XB and UNLOCK_INDEX_XB routines, as well as the data ones, also act upon this lock count. Therefore, you can lock a file 100 times in a row, but only on the first lock are any operations actually performed, and only on the last unlock are any performed. Other lock/unlock calls (other than the first lock or last unlock) simply increment or decrement the lock count for that handle.

CHECK_REMOTE_XB

Uses REMOTEPACK

```
   IN                        OUT
RP.func                   RP.stat
RP.handle                 RP.isRemote
  -or-                    RP.flags=0
RP.drive                  RP.isShare=1
```

If RP.handle is non-zero, determine if the specified handle of a file or device is remote. If the handle is local (e.g., not a network file or device), RP.isRemote returns 0, otherwise it is remote. RP.flags=0 and RP.isShare=1 on return for either a handle or drive check under OS/2.

If RP.handle is zero, determine if the drive specified in RP.drive is remote. Drive A: is 1, B: is 2, C: is 3, and so on. To check the default drive use 0 (the default drive is the current drive). If the drive is local (e.g., not a network drive), RP.isRemote returns 0, otherwise it is remote.

The significance of the remote-ness is less important in a multitasking environment since sharing of resources (files, in particular) must always be managed, compared to single-tasking environments where, typically, sharing (locking mechanisms) need only be performed when the resource is able to be accessed by another process (i.e. is a 'network' drive). Note that the resource need not be located elsewhere to be classified as remote: Drives or devices or files on the same machine may be classified as remote if the network software is redirecting local access (such as on a server).

Note: This routine is not mutex-protected.

RELOCK_XB

Uses LOCKPACK

IN	OUT
`LP.func`	`LP.stat`
`LP.handle`	
`LP.xlMode`	
`LP.dlMode`	
`LP.recStart=0`	
`LP.nextPtr`	

Relock all bytes of the index files for the mode specified in LP.xlMode (index files) and LP.dlMode (data files). Also relock all bytes in the related data file. LP.recStart must be 0 for each pack.

Set LP.xlMode=1 to select a shared lock for the index file; set to 0 for an exclusive lock. Set LP.dlMode=1 to select a shared lock for the data file; set to 0 for an exclusive lock.

If the lock mode is from exclusive to shared, the file is flushed before the shared state is set. BULLET maintains the current lock state and knows which direction the lock is going in. The lock state (shared or exclusive) can be determined by the SIP and SDP structures from the STAT_XB routines. This routine does not affect the lock count, nor are the headers reloaded (nor should they be).

The lock state is on a file handle basis, not on an LP[] pack basis. This means the file, as identified by the handle, is in the lock state last set.

Note: The lock switch is made atomic: Rather than unlocking, and then locking again in the new state, this performs all operations without the possibility that another process can grab the lock away, if supported by the OS. If relock is not supported by the OS, and shared (read-only) locks are supported, the relock will be performed by Bullet rather than the OS. If Bullet performs the relock, another, non-Bullet process may be able to steal away the lock since it is no longer guaranteed to be atomic.

RELOCK_INDEX_XB

Uses LOCKPACK

IN	OUT
`LP.func`	`LP.stat`
`LP.handle`	
`LP.xlMode`	

Relock all bytes of the index file for the mode specified in LP.xlMode.

Set LP.xlMode=1 to select a shared lock; set to 0 for an exclusive lock. If the lock mode is from exclusive to shared, the file is flushed before the shared state is set. BULLET maintains the current lock state and knows which direction the lock is going in. The lock state (shared or exclusive) can be determined by the SIP structure from the STAT_INDEX_XB routine. This routine does not affect the lock count, nor is the header reloaded (nor should it).

Note: The lock switch is made atomic: Rather than unlocking, and then locking again in the new state, this performs all operations without the possibility that another process can grab the lock away, if supported by the OS. If relock is not supported by the OS, and shared (read-only) locks are supported, the relock will be performed by Bullet rather than the OS. If Bullet performs the relock, another, non-Bullet process may be able to steal away the lock since it is no longer guaranteed to be atomic.

RELOCK_DATA_XB

Uses LOCKPACK

IN	OUT
`LP.func`	`LP.stat`
`LP.handle`	
`LP.dlMode`	
`LP.recStart`	
`LP.recCount`	

Relock all bytes of the data file for the mode specified in LP.dlMode.

If the lock mode is from exclusive to shared, the file is flushed before the shared state is set. BULLET maintains the current lock state and knows which direction the lock is going in. The lock state (shared or exclusive) can be determined by the SDP structure from the STAT_DATA_XB routine. This routine does not affect the lock count , nor is the header reloaded (nor should it be).

You must set LP.recStart=0 to lock all bytes. To lock a single record, or set of contiguous records, set LP.recStart= record# to relock and LP.recCount to the number of records to relock. Multiple single records are allowed, but each must then be unlocked individually, in the same format (start, count) as the lock.

Note: The lock switch is made atomic: Rather than unlocking, and then locking again in the new state, this performs all operations without the possibility that another process can grab the lock away, if supported by the OS. If relock is not supported by the OS, and shared (read-only) locks are supported, the relock will be performed by Bullet rather than the OS. If Bullet performs the relock, another, non-Bullet process may be able to steal away the lock since it is no longer guaranteed to be atomic.

CP Level

- DELETE_FILE_DOS
- RENAME_FILE_DOS
- CREATE_FILE_DOS
- OPEN_FILE_DOS
- SEEK_FILE_DOS
- READ_FILE_DOS
- WRITE_FILE_DOS
- CLOSE_FILE_DOS
- ACCESS_FILE_DOS
- EXPAND_FILE_DOS
- MAKE_DIR_DOS
- COMMIT_FILE_DOS

DELETE_FILE_DOS

Uses DOSFILEPACK

IN	OUT
DFP.func	DFP.stat
DFP.filenamePtr	

Delete the specified file.

Note: In OS/2, DosForceDelete is used so the file is not recoverable with the UNDELETE command.

RENAME_FILE_DOS

Uses DOSFILEPACK

IN	OUT
DFP.func	DFP.stat
DFP.filenamePtr	
DFP.newNamePtr	

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

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 /LP100/PROJ94A.INF and the newFilenamePtr filename is /ARCH/PROJ93A.INA, the file is essentially renamed and also moved to the /ARCH directory.

CREATE_FILE_DOS

Uses DOSFILEPACK

IN	OUT
DFP.func	DFP.stat
DFP.filenamePtr	
DFP.attr	

Create a new file.

The specified filename/pathname must *not* already exist.

The file created is not left open. You must OPEN_FILE_DOS 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
SubDir	10h	FILENAME is a subdirectory
Archive	20h	file is marked for archiving

Note: Use MAKE_DIR_DOS to create a subdirectory.

OPEN_FILE_DOS

Uses DOSFILEPACK

IN	OUT
DFP.func	DFP.stat
DFP.filenamePtr	DFP.handle
DFP.asMode	

Open the file with the access-sharing mode, returning the handle on success.

SEEK_FILE_DOS

Uses DOSFILEPACK

IN	OUT
DFP.func	DFP.stat
DFP.handle	DFP.seekTo
DFP.seekTo	
DFP.method	

Position the file pointer of the file to the seekTo position based on the method specified.

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 last byte of a sector (512th byte, but offset 511), set the offset value to 511 and use Method 0. On return, the absolute offset value of the new position is returned. This return value is useful with Method 2 since you can specify an offset of 0 and have the file length returned. To move to the start of the file, use method 0, offset 0. To move to the first byte of the second sector, use offset 512.

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

READ_FILE_DOS

Uses DOSFILEPACK

IN	OUT
DFP.func	DFP.stat
DFP.handle	DFP.bytes
DFP.bytes	
DFP.bufferPtr	

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

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 EOF was reached during the read; if 0, then was already at EOF.

WRITE_FILE_DOS

Uses DOSFILEPACK

IN	OUT
DFP.func	DFP.stat
DFP.handle	DFP.bytes
DFP.bytes	
DFP.bufferPtr	

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

If the number of bytes written is less than the specified bytes, this routine returns an error. 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.

***Note*: <u>If the specified bytes to write is 0, the file is truncated at the current file pointer position</u>.**

CLOSE_FILE_DOS

Uses DOSFILEPACK

IN	OUT
`DFP.func`	`DFP.stat`
`DFP.handle`	

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

ACCESS_FILE_DOS

Uses DOSFILEPACK

IN	OUT
`DFP.func`	`DFP.stat`
`DFP.filenamePtr`	
`DFP.asMode`	

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

Basically, a Does-File-Exist routine. It uses the specified access-sharing mode when trying to open the file. For example, if you specify DFP.attr = 0x0042 (R/W access + Deny None sharing) and issue ACCESS_FILE_DOS on a Read-Only file, an error is returned. A **sharing mode** must be specified; it cannot be left 0.

EXPAND_FILE_DOS

Uses DOSFILEPACK

IN	OUT
`DFP.func`	`DFP.stat`
`DFP.handle`	
`DFP.bytes`	

Expands the file by the number of bytes beyond its current size.

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.

Note: The file space is not initialized.

MAKE_DIR_DOS

Uses DOSFILEPACK

IN	OUT
`DFP.func`	`DFP.stat`
`DFP.filenamePtr`	

Create a new subdirectory.

COMMIT_FILE_DOS

Uses DOSFILEPACK

IN	OUT
`DFP.func`	`DFP.stat`
`DFP.handle`	

Flushes the OS system buffers for the handle, and updates the directory entry for size.

Bullet Error Codes

Bullet error codes are numbered so that they do not overlap OS error codes. The first general Bullet error number is 8193, but Bullet also returns select system error code numbers instead of duplicating system codes (those listed below less than 8192). In addition, if the error is reported by the OS (for example, attempting to access a locked file), then the OS error in returned by Bullet. For a list of OS errors, see OS/2 Dos API Errors in the online manual, or refer to the OS documentation.

System/General Error Codes

8 EXB_NOT_ENOUGH_MEMORY
cannot get memory requested

15 EXB_INVALID_DRIVE
not a valid drive letter

38 EXB_UNEXPECTED_EOF
unexpected end-of-file where bytes requested for read exceeded EOF

39 EXB_DISK_FULL
disk full on WriteFile

80 EXB_FILE_EXISTS
cannot create file since it already exists

105 EXB_SEM_OWNER_DIED
owner of mutex semaphore died (used in place of Win32 80h)

640 EXB_TIMEOUT
mutex semaphore timed out (used in place of Win32 102h)

8192+ EXB_OR_WITH_FAULTS
1=flush failed on handle close
2=free memory failed on handle close
4=failed memo handle close (data handle only)

During a CLOSE_XB routine, the close process continues regardless of errors, and so the errors are accumulated. For example, 8193 means the flush failed, and 8195 means both the flush and the free failed (8192+1+2=8195). If the error occurred in the actual DosClose() API call, only that error is returned (it will be an OS error code).

8251 EXB_216501
INT21/6501h not supported by DOS extender (see ccdosfn.c)

8256 EXB_216506
INT21/6506h not supported by DOS extender (see ccdosfn.c)

8300 EXB_ILLEGAL_CMD
function not allowed

8301 EXB_OLD_DOS

OS version < MIN_DOS_NEEDED

8302 EXB_NOT_INITIALIZED
init not active, must do INIT_XB before using Bullet

8303 EXB_ALREADY_INITIALIZED
init already active, must do EXIT_XB first

8304 EXB_TOO_MANY_HANDLES
more than 1024 opens requested, or more than license permits (100, 250, 1024)

8305 EXB_SYSTEM_HANDLE
Bullet won't use or close handles 0-2

8306 EXB_FILE_NOT_OPEN
the handle is not a Bullet handle, including the handle supplied in OP.xbLink

8307 EXB_FILE_IS_DIRTY
tried to reload header but current still dirty; flush the file before reloading the header

8308 EXB_BAD_FILETYPE
attempted to do a key file operation on non-key file, or a data operation on a non-data file

8309 EXB_TOO_MANY_PACKS
too many INSERT, UPDATE, REINDEX, LOCK_XB packs (more than 256)

8310 EXB_NULL_RECPTR
null record pointer passed to Bullet (.recPtr==NULL)

8311 EXB_NULL_KEYPTR
null key pointer passed to Bullet (.keyPtr==NULL)

8312 EXB_NULL_MEMOPTR
null memo pointer passed to Bullet (.memoPtr==NULL)

8313 EXB_EXPIRED
evaluation time period has expired, reinstall if time remaining

8314 EXB_BAD_INDEX
Query/SetSysVars index selection is beyond the last one

8315 EXB_RO_INDEX
SetSysVars index item is read-only

8316 EXB_FILE_BOUNDS
file size > 4GB, or greater than the SetSysVars value

8397 EXB_FORCE_ROLLBACK
rollback test requested (last AP[].nextPtr=-1 for Insert/Update) (test-use only)

8398 EXB_INVALID_DLL
DLL seems to be invalid (8399 similar)

Multi-access Error Codes

8401 EXB_BAD_LOCK_MODE
lock mode (LP) not valid, must be 0 or 1

8402 EXB_NOTHING_TO_RELOCK
cannot relock without existing full-lock

8403 ERR_SHARED_LOCK_ON
unlikely error, write access needed for flush, but lock is shared

Index Error Codes

8501 EXB_KEY_NOT_FOUND
exact match of key not found

8502 EXB_KEY_EXISTS
key exists already and dups not allowed

8503 EXB_END_OF_FILE
already at last index order

8504 EXB_TOP_OF_FILE
already at first index order

8505 EXB_EMPTY_FILE
nothing to do since no keys

8506 EXB_CANNOT_GET_LAST
cannot locate last key

8507 EXB_BAD_INDEX_STACK
index file is corrupt

8508 EXB_BAD_INDEX_READ0
index file is corrupt

8509 EXB_BAD_INDEX_WRITE0
index file is corrupt

8521 EXB_OLD_INDEX
incompatible Bullet index, use ReindexOld subroutine, if available

8522 EXB_UNKNOWN_INDEX
not a Bullet index file

8523 EXB_KEY_TOO_LONG
keylength > 62 (or 64 if unique), or is 0

8531 EXB_PARSER_NULL
parser function pointer is NULL

8532 EXB_BUILDER_NULL
build key function pointer is NULL

8533 EXB_BAD_SORT_FUNC
CIP.sortFunction not valid (not 1-6, or a custom sort-compare)

8534 EXB_BAD_NODE_SIZE
CIP.nodeSize is not 512, 1024, or 2048

8535 EXB_FILENAME_TOO_LONG
CIP.filenamePtr->pathname greater than file system allows

This error is detected only for the file system installed, and does not detect using pathnames greater than 80 on a FAT system if HPFS, NTFS, VFAT is installed. The OS returns its own error in this case, after the fact.

8541 EXB_KEYX_NULL
key expression is effectively NULL

8542 EXB_KEYX_TOO_LONG
CIP.keyExpPtr->expression is greater than 159 bytes

8543 EXB_KEYX_SYM_TOO_LONG
fieldname/funcname in expression is longer than 10 chars

8544 EXB_KEYX_SYM_UNKNOWN
fieldname/funcname in expression is unknown or misspelled

8545 EXB_KEYX_TOO_MANY_SYMS
too many symbols/fields used in expression (16 max)

8546 EXB_KEYX_BAD_SUBSTR
invalid SUBSTR() operand in expression

8547 EXB_KEYX_BAD_SUBSTR_SZ
SUBSTR() exceeds field's size

8548 EXB_KEYX_BAD_FORM
didn't match expected symbol in expression (missing paren, etc.)

8551 EXB_NO_READS_FOR_RUN
unlikely error, use different reindex buffer size to fix

8552 EXB_TOO_MANY_RUNS
unlikely error, too many runs (64K or more runs)

8553 EXB_TOO_MANY_RUNS_FOR_BUFFER
unlikely error, too many runs for run buffer

8554 EXB_TOO_MANY_DUPLICATES
more than 64K "identical" keys since the last enumerator used was 0xFFFF — if ever you have this error, REINDEX_XB should be used to resequence the enumerators

8561 EXB_INSERT_RECNO_BAD
AP.recNo cannot be > 0 if inserting with INSERT_XB

8562 EXB_PREV_APPEND_EMPTY
no previous append for INSERT_XB yet AP.recNo==0x80000000

8563 EXB_PREV_APPEND_MISMATCH
previous append's xbLink does not match this — if this pack's AP.recNo=0x80000000 then this pack's AP.handle must be the same handle as that of the last pack that added a record

8564 EXB_INSERT_KBO_FAILED
could not back out key at INSERT_XB

8565 EXB_INSERT_DBO_FAILED
could not back out data records at INSERT_XB

8571 WRN_NOTHING_TO_UPDATE
all AP.recNo=0 at UPDATE_XB so nothing to do

8572 EXB_INTERNAL_UPDATE
internal error UPDATE_XB, not in handle/record# list

8573 EXB_FAILED_DATA_RESTORE
could not restore original data record (*)

8574 EXB_FAILED_KEY_DELETE
could not remove new key (*)

8575 EXB_FAILED_KEY_RESTORE
could not restore original key(*)

(*) original error, which forced a back-out, has been replaced by this error — this error is always returned in the first AP.stat (-1 on data, 1 on index)

Data Error Codes

8601 EXB_EXT_XBLINK
xbLink handle is not an internal DBF, as was specified during the index file's creation - the Bullet routine called requires a Bullet DBF data file (instead use index-only access methods like NEXT_KEY_XB).

8602 EXB_FIELDNAME_TOO_LONG
fieldname is > 10 characters

8603 EXB_RECORD_TOO_LONG
record length is > 64K

8604 EXB_FIELD_NOT_FOUND
fieldname not found in descriptor info

8605 EXB_BAD_FIELD_COUNT
fields <= 0 or >= MAX_FIELDS; also use of a field number which is beyond the last field

8606 EXB_BAD_HEADER
bad header (reclen=0, etc.)

8607 EXB_BUFFER_TOO_SMALL
buffer too small (pack buffer < record length)

8608 EXB_INTERNAL_PACK
internal error in PackRecords

8609 EXB_BAD_RECNO
record number=0 or > records in data file header, or pack attempt on empty data file

8610 WRN_RECORD_TAGGED
record's tag field matches skip tag

Memo Error Codes

8701 WRN_CANNOT_OPEN_MEMO
the DBF header has bits 3 & 7 set, which indicates that a memo file is attached to this DBF, but the DBT memo file failed to open — the DBF open continues, with this warning code returned

8702 EXB_MEMO_NOT_OPEN
no open memo file for operation

8703 EXB_BAD_BLOCKSIZE
memo blocksize must be at least 24 bytes

8704 EXB_MEMO_DELETED
memo is deleted

8705 EXB_MEMO_PAST_END
memo data requested is past end of record

8706 EXB_BAD_MEMONO
memo number is not valid

8707 EXB_MEMO_IN_USE
memo add encountered likely corrupt memo file — avail list indicates this memo record is deleted, but the memoAvail link for the memo indicates it is use (memoAvail link==0x8FFFF)

8708 EXB_BAD_AVAIL_LINK
memo avail link cannot be valid (e.g., memoAvail==0)

8709 EXB_MEMO_ZERO_SIZE
memo data has no size (size is 0)

8710 EXB_MEMO_IS_SMALLER
memo attempt to shrink but memo size is already <= size requested

Specifications

- DOS OS Calls Made
- OS/2 API Calls Made
- Win32 API Calls Made
- Bullet Memory Usage
- IX3 File Format
- DBF File Format
- DBT File Format
- Custom Sort-Compare Function
- Custom Build-Key
- Custom Expression Parser

DOS API Calls Made

The following are the API calls made by Bullet.

INT21/3E	Close File	INT21/4409	Drive Remote
INT21/39	Create Directory	INT21/440A	File Remote
INT21/5B	Create File	INT21/6506	Get Sort Table
INT21/41	Delete File	INT21/6501	Get CountryCode/Codepage Info
INT21/56	Rename File	INT21/59	Get Extended Error
INT21/3D	Open File	INT21/30	Get Version
INT21/3F	Read File	INT21/67	Set Handle Count
INT21/42	Seek File	INT21/2A/2C	Get Time
INT21/45	Update Directory Entry	INT21/6520	Map Character
INT21/40	Write File	INT2F/1000	Share Check
INT21/5C	Lock File	INT31/6	DPMI, Get Segment Base Address

Memory allocation routines are defined in ccdosfn.c.

OS/2 API Calls Made

The following are the API calls made by Bullet.

DosAllocMem	DosClose	DosCopy
DosCloseMutexSem	DosCreateDir	DosCreateMutexSem
DosDelete (*)	DosErrClass	DosExitList
DosForceDelete	DosFreeMem	DosGetDateTime
DosMapCase	DosOpen	DosQueryCollate
DosQueryCp	DosQueryCtryInfo	DosQueryCurrentDisk
DosQueryFSAttach	DosQueryHType	DosQuerySysInfo
DosMove	DosRead	DosReleaseMutexSem
DosRequestMutexSem	DosResetBuffer	DosScanEnv
DosSetFileLocks	DosSetFilePtr	DosSetFileSize
DosSetMaxFH	DosSetRelMaxFH	DosWrite

(*) DosForceDelete is used in favor of DosDelete.

WIN32 API Calls Made

CharToOemBuffA	GetLocalTime	OemToCharBuffA	OpenFile
CharUpperA	GetLocaleInfoA	MoveFileA	ReadFile
CharUpperBuffA	GetVersion	CloseHandle	ReleaseMutex
CreateDirectoryA	GetVersionExA	FlushFileBuffers	SetEndOfFile
CreateFileA	GetTickCount	GetACP	SetFilePointer
CreateMutexA	GlobalAlloc	GetOEMCP	SetHandleCount
DeleteFileA	GlobalFree	GetSystemDefaultLCID	UnlockFile
GetDriveTypeA	GlobalMemoryStatus	GetFileSize	UnlockFileEx
GetEnvironmentVariableA		LockFile	GetFileType
WaitForSingleObject	GetVolumeInformationA	LockFileEx	GetLastError
WriteFile			

Bullet Memory Usage

Memory is committed when allocated, using the PAG_COMMIT and the PAG_WRITE flags. This is memory allocated by Bullet itself. Additional memory needs are made by your code, such as parameter pack data, key buffers, and data record buffers. This applies to the default memory allocation scheme used by Bullet, and will vary somewhat if SET_VECTORS_XB with VECTOR_MALLOC is used.

Code

Bullet uses 8 pages for code, or less than 32KB.

Data

o **Shared Data**

A single page of shared memory is used by all Bullet processes.

o **Instance Data**

One page of private memory is used by each Bullet processes.

o **Handle Data**

One page of private memory is used by each open Bullet index file. For open data files, one page of private memory is used for files with 121 or fewer fields. Two pages are used for files with 249 or fewer fields. Thereafter, one page for each additional 128 fields, up to 1024 max fields.

For example, if one machine is running 2 Bullet processes, each with 10 open data files with 12 fields each, and 10 index files (one for each data file), its total memory usage is:

Total code is 32KB.
Shared data is 4KB.
Instance data is 2 processes * 4KB, or 8KB (plus INIT_XB use shown below).
DBF file data is 2 * 10 files * 4KB, or 80KB.
Index file data is 2 * 10 files * 4KB, or 80KB.

Total memory committed by Bullet for the above is 200KB, plus code and data of your two applications (or your single application, if the same application is being run twice). With no files open, for example when starting your program, only 40KB is committed. Thereafter, 4KB per open file. The memory is freed when the file is closed.

o **Temporary Data**

Additional memory is allocated on a temporary basis, where the allocation is requested on entry to, and is freed upon exiting from, the routine called. INIT_XB's allocation can be considered permanent since its allocations are not released until the program has ended, or EXIT_XB is issued. The following are the routines and the single requested amount:

Routine	Memory Allocated, in KB
INIT_XB	8 for 1024 MAX_FILES version; 4KB for 100 and 250 MAX_FILES versions
BACKUP_FILE_XB	8
CREATE_DATA_XB	4 for 1-121 fields, 8 for 122-249 fields, ...
CREATE_INDEX_XB	4
PACK_RECORDS_XB	adjustable, 128 default (less if file is smaller)
REINDEX_XB	adjustable, 144 default (minimum size is 48KB)
UPDATE_XB	varies: 40 + sum of record lengths where AP[].recNo!=0

o **Stack Data**

Stack requirements are 8KB minimum for Bullet. No single stack allocation requests more than 4KB at a time (*i.e.* all changes to ESP (the CPU stack pointer) are less than 4KB at any one time), but some routines nest and require up to the minimum 8KB in total. The *minimum* recommended stack size for your Bullet application is 16KB. It's likely that you need to use a much larger size for your main program's stack use. If you have any doubt about stack space, double it, twice even.

IX3 File Format

The IX3 index file is composed of a header followed by node data. The header layout is detailed below, followed by the node format.

Index Header

```
// nnn, where nnn is the offset of that item relative to the start of the file

CHAR fileID[4];      // 000 file id = '31ch'
ULONG nodeSize;      // 004 size of a node, in bytes
ULONG rootNode;      // 008 root node (1-based)
ULONG noKeys;        // 012 total number of keys
ULONG availNode;     // 016 next available node (link to, 0 if none, 1-based)
ULONG freeNode;      // 020 next free node
ULONG keyLength;     // 024 length of key
ULONG maxKeys;       // 028 maximum number of keys on a node
ULONG codePage;      // 032 code page from CreateIndexFile
ULONG countryCode;   // 036 country code from CreateIndexFile
ULONG sortFunction;  // 040 system (1-9) or custom (10-19)
                     //     high word has flags: bit0=1 dups allowed
                     //                          bit1-15 rez

// Translated key expression as done by Parser during CreateIndex and Reindex.
// More details on this is in the Custom Expression Parser Specifications.
```

```
// For each key part in KH.expression a 4-byte structure is used in XLATEX:

typedef struct _XLATEX {
 CHAR  ftype;   // field type (C,N,L, etc.),if bit7=1 and C then do UPPER key
 CHAR  length;  // bytes to use starting at offset (never > 64)
 SHORT offset;  // byte offset into data record that length bytes are to be used
} XLATEX;

ULONG xlateCount;          // 044 number of key fields (64/4=16 max fields)
XLATEX xlateExpression[16]; // 048 key construct info (16 dword's worth)
CHAR  miscWorkspace[236]; // 112-347 B-tree workspace
CHAR  expression[160];    // 348 key expression, user (0-Terminated)
ULONG CTsize;             // 508 size of collate table following
CHAR  collateTable[256];  // 512 collate table, fill at CreateIndexXB
CHAR  rez[256];           // 768 to 1023 reserved (header size=1024 bytes)
```

Node Data

Directly after the header the node data starts. Each node is either 512, 1024, or 2048 bytes long. Each node contains a key count, indicating the number of active keys on the node, followed by key data.

```
// nnn, where nnn is the offset of that item relative to the start of the node

CHAR  keyCount;         // 000 1 to maxKeys (in header above)
ULONG backNode;         // 001 previous node page, or 0 if this node is a leaf
XNODE node[maxKeys];    // 005...
```

For each key on the node:

```
typedef struct _XNODE {
CHAR  keyValue[keyLength]; // 005   actual key   (keyLength from header)
ULONG recordNo;            // 005+keyLength   record number for key
ULONG fwdNode;             // 005+keyLength+4   next node page, or 0 if leaf
} XNODE;
```

backNode and fwdNode are node numbers. The first node is 1, and is located directly after the header. The last node used is at header:freeNode-1. Each fwdNode of a key is also the next key's backNode. If the node has had all keys removed, its node number is placed on the top of the header:availNode list, and the first 4 bytes of the node are used as a link to the previous list top.

DBF File Format

The DBF data file is composed of a header, field descriptors, one per field, and the actual record data. The header layout is detailed below, followed by the field descriptor layout and then the description of the data record.

DBF Header

```
// nnn, where nnn is the offset of that item relative to the start of the file

CHAR  fileID;           // 000 file id byte
CHAR  lastUpdateYR;     // 001 binary year-1900
CHAR  lastUpdateMO;     // 002 binary month (1-12)
CHAR  lastUpdateDA;     // 003 binary day (1-31)
ULONG noRecords;        // 004 total number of records
SHORT headerLength;     // 008 length of data header
SHORT recordLength;     // 010 record length
SHORT nada;             // 012 reserved
CHAR  xactionFlag;      // 014 flag indicating incomplete dBASE transaction (n/a)
CHAR  encryptFlag;      // 015 flag indicating encrypted (n/a)
CHAR  filler[16];       // 016 fill to 32 bytes
```

Field Descriptors

For each field, a descriptor is stored in the DBF. The first descriptor starts directly after the header, at file offset 32 (the 33rd byte). Each descriptor is 32 bytes. After the last descriptor, a byte with ASCII value 13 (0x0D) is stored. Following this byte, the record data starts.

```
// nnn, where nnn is offset of item relative to the start of the descriptor

CHAR  fieldName[11];    // 000 ASCII, UPPER, underscore, zero-filled, (0T)
CHAR  fieldType;        // 011 UPPER C,N,D,L,M
ULONG fieldDA;          // 012 not used
CHAR  fieldLength;      // 016 1-255 bytes, depending on fieldType
CHAR  fieldDC;          // 017 places right of decimal point
SHORT altFieldLength;   // 018 alternate field length when fieldLength==0
CHAR  filler[12];       // 020 not used

// altFieldLength is proprietary to Bullet, and is not standard Xbase.
// To use it, set fieldLength=0 and altFieldLength to > 255 bytes.
```

Record Data

The DBF data are free-form, fixed-length records. Each data record starts with a one-byte 'tag' field, which is implicitly defined for all records (hence, it is not a formal field and has no descriptor). Following the tag field is the first field of the record, and following that field (whose length is described in the field's descriptor) is the next field, and so on. No separators are used between fields. After the very last data record in the file, DBF specification dictates that an *end of file* marker be placed, so at the end of the file is a byte of value ASCII 26 (0x1A).

```
Record layout is as you define in your application.  It must match the layout
as described in the field descriptors, byte-for-byte.
```

Increasing DBF Performance

Records are stored in the order they were written. To improve performance, especially indexed-sequential access, the data file may be sorted, or clustered, by reading each record in primary key order, then writing that record to a new DBF data file. Repeat for each record. After all records have been written, reindex the newly created DBF data file (and all related index files). After this, delete the old files (data and index), and rename the new ones to the filenames required. This technique maximizes cache efficiency, and can easily offer 10x performance increase in access speed.

DBT File Format

The DBT memo file is composed of a header followed by memo data, stored in one or more blocks. The header layout is detailed below, followed by the memo record.

DBT Header

```
// nnn, where nnn is the offset of that item relative to the start of the file

ULONG memoAvailBlock;   // 000 next available block (header is block 0)
ULONG memoRez;          // 004 not used
CHAR  memoFilename[8];  // 008 filename proper (first 8 of filename proper)
ULONG memoRez2;         // 016 not used (apparently)
ULONG memoBlockSize;    // 020 block size, must be at least 24

// the rest of the header block (to block size bytes) is unused
```

Memo Record

```
// nnn, where nnn is offset of item relative to the start of the memo record

ULONG memoAvail;        // 000 next available link
ULONG memoSize;         // 004 size of data (including this and memoAvail)
CHAR  memoData;         // 008 for as many bytes as memoSize, less 8
```

A memo may use one or more blocks (each block is a fixed size), but allocations are always contiguous. Unused bytes after the memo data (to the end of the last block allocated to that memo record) are undefined. memoAvail is 0x8FFFF for all active memo records. For deleted memo records, memoAvail is used as a link in the memoAvail list. memoSize is the total bytes used by the memo, including the memoAvail and memoSize data, so it is the size of the real data + 8 bytes.

Removing Memo Records

While the memo file used by Bullet does reuse deleted memo space, there may be a need to shrink a memo file by removing all inactive memo data. A similar technique to that above could be used to remove memo records from a DBT. Instead of using PackRecords_XB, read each DBF record in sequence, and if not to be deleted, write that record to a new DBF data file and its memo records to a new DBT memo file. If the record read is to be deleted, skip it (and its memo record(s)) and continue with the next DBF data record. Repeat while more records. Delete or archive the old DBF/DBT pair, and use the new files in their place. Reindex. Also see `compact.c` on the distribution disk for an example technique of compacting a database.

Custom Sort-Compare Function

Bullet provides 10 custom sort-compare functions, in addition to the 6 intrinsic sort-compare functions (ASCII, NLS, and the four integer compares). The custom function you supply is not actually a sort function, as the name implies, but a compare function. Basically, two strings are supplied and your function determines string1's relation to string2 (<, >, or ==).

The strings supplied (via pointers) are not C strings, and they are not (necessarily) 0-terminated. A count value is passed, indicating the number of bytes to compare. The handle of the index file for which this compare is being done is also supplied, so that you can interrogate the index file state (STAT_INDEX_XB) for any additional information required.

In addition to the compare function this routine performs, a special-case call is made to this routine requesting a pointer to a string of HIGH-VALUES for this sort compare. The pointer must be to a static memory area that exists for as long as the index file is open, and must be *at least* as long as count. This special-case call is indicated by both string pointers==NULL.

To use a custom sort-compare function, first use SET_SYSVARS_XB to assign the custom sort ID (10 to 19) with the function's address pointer. Once assigned, an index file may be created with its CIP.sortFunction set to the sort ID (10-19). Also, any previously created index file with a custom sort ID may now be opened (but only after you used SET_SYSVARS_XB to assign the custom sort-compare function pointer). During the index file create, the sort ID you specified for the create is stored in the index file. When that index file is later opened, that same sort ID is used, and so requires that the custom sort-compare function already be assigned (with SET_SYSVARS_XB) before opening the index file. This means that you need to be consistent in your custom sort ID numbering, since each index created forever uses that sort ID you specified.

It's simple to create a custom sort-compare function. The calling convention is APIENTRY (DOSX32 = __cdecl, OS/2= _syscall, Win32=_stdcall), and the parameters are passed to your function on the stack (by Bullet). A sample prototype for a custom sort-compare function follows:

```
LONG APIENTRY YourCustomSort10(PVOID str1,
                               PVOID str2,
                               ULONG count,
                               ULONG handle);
```

If the pointers are not NULL, your routine is to compare str1 to str2, for count bytes, and is to return:

```
    -1 if str1 is less than str2
     0 if str1 is equal to str2
     1 if str1 is greater than str2
```

str is not a C string; it can be anything. Cast as required, depending on the data expected.

If str1 and str2 are both NULL, your routine must return a pointer to a static object that contains HIGH-VALUES for the object type. For example, if the sort-compare is for IEEE floating-point, then the function is to return a pointer to a static data area filled with the highest floating-point value. Depending on your sort-compare routine's functionality, you may need just a single high-value, or multiple high-values, one after the other (e.g., if you are supporting compound key values for binary keys). The count parameter indicates the total bytes needed, so divide by the object size to get the number of objects required. Be aware that the object size (in count) is +2 bytes for the enumerator if DUPS_ALLOWED was specified when the index file was created. This high-values object is used in the REINDEX_XB routine, and also the LAST_KEY_XB and GET_LAST_XB routines.

Custom Build-Key

Bullet provides an internal build-key routine that constructs the key from the data record supplied. The internal routine can be overloaded by your custom build-key routine if you need additional functionality. It may be used in conjunction with a custom sort-compare function, or an intrinsic Bullet sort-compare.

Developing a custom build-key routine requires delving into the internal Bullet data structures. It is more complicated than a custom sort-compare function, but not really any more complex. The handle of the index file is passed, and using this, STAT_INDEX_XB is called to get the SIP.herePtr pointer. This is the pointer to the internal Bullet data structure for this index file. What needs to be accessed in this structure is the translated key expression. From this, you have the starting offset in the data record, and the byte count to use, for each key component (up to 16 components per key). The offset value as stored in the XLATEX structure does not include the tag field byte. Therefore, to locate to the correct offset, add 1 to the value in offset. For example, XLATEX.offset=0 means to use the first field, which is the first byte *after* the tag field byte, but the physical offset, as referenced to recPtr, is not at offset=0, but is at offset=1.

This translated key expression structure is:

```
// (This is an excerpt from the IX3 header format)

// Translated key expression as done by Parser during CreateIndex and Reindex.
// More details on this is in the Custom Expression Parser Specifications.
// For each key part in KH.expression a 4-byte structure is used:

typedef struct _XLATEX {
 CHAR  ftype;   // field type (C,N,L, etc.),if bit7=1 and C then do UPPER key
 CHAR  length;  // bytes to use starting at offset (never > 64)
 SHORT offset;  // byte offset into data record that length bytes are to be used
} XLATEX;       // (note that offset does not count tag field byte)

ULONG xlateCount;           // 044 number of key fields (64/4=16 max fields)
XLATEX xlateExpression[16]; // 048 key construct info (16 dword's worth)
```

xlateExpression is at offset +48 relative the IX3 index header. However, SIP.herePtr points to -384 relative the IX3 index header start. Therefore, to locate to xlateExpression, you must add 384 to 48. This means that xlateExpression[0].ftype is located at SIP.herePtr+432. The number of valid key components in xlateExpression is stored in xlateCount (at SIP.herePtr+428).

The calling convention for your custom build-key function is APIENTRY, and the parameters are passed to your function on the stack (by Bullet). A sample prototype for a build-key function follows:

```
ULONG APIENTRY YourBuildKey(ULONG handle,
                            PVOID recordPtr,
                            PVOID keyPtr,
                            PULONG keyLenPtr
                            PULONG sortFuncPtr);
```

Using the data from xlateExpression, you are to build a key from the data record located at the passed pointer, recordPtr, and are store the built key in the buffer located at keyPtr. For each key component, you copy from the data record xlateExpression[].length bytes starting at xlateExpression[].offset+1 (given the 1-byte tag field which is not accounted for otherwise), and build other key components after previously build parts. If the index file allows duplicate keys (DUPS_ALLOWED is flagged in SIP.sortFunction), then append an enumerator to the end of the key proper. The handle of the index file is passed, which is used when calling STAT_INDEX_XB (to get SIP.herePtr). The return is 0 if successful, or an appropriate Bullet error code (EXB_) should be used. In addition, the key length is placed in the ULONG data pointed to by keyLenPtr (SIP.keyLength may be used), and the sort-compare function is placed in the ULONG data pointed to by sortFuncPtr (SIP.sortFunction may be used).

The routine is also to check if the tag field of the data record matches the skip tag value as set by SET_SYSVARS_XB. If the tag field matches, WRN_SKIP_KEY is to be returned as the 'error' code. The key is built regardless of a match.

Custom Expression Parser

Bullet provides an internal key expression parser routine that constructs the translated key expression stored in the index file header. The internal routine can be overloaded by your custom expression parser routine if you need additional functionality. It may be used in conjunction with a custom sort-compare function, with a custom build-key routine, or with an intrinsic Bullet sort-compare.

Developing a custom expression parser routine requires delving into the internal Bullet data structures. It is more complicated than a custom sort function, and it is also much more complex. Unlike the custom sort-compare and build-key functions, no handle is passed to the parser. This is because, rather than using the handle to get the SIP.herePtr, this pointer is passed directly to this routine. This is the pointer to the internal Bullet data structure for this index file. What needs to be accessed in this structure is the translated key expression location, as well as the text version of the key expression, as supplied by the programmer/user. To the XLATEX data you place the starting offset in the data record, and the byte count to use, for each key component you parse from the key expression (up to 16 components per key). The offset value as stored in the XLATEX structure does not include the tag field byte. Therefore, the correct offset to store is the physical offset within the record, minus 1. For example, XLATEX.offset=0 should be used for the offset of the first field, which is the first byte after the tag field byte. For each component parsed, an XLATEX data structure is added to the xlateExpression data area (up to 16). Unused XLATEX components must be set to 0. When all components have been stored, the xlateCount value is set to the number of key components stored.

DHDptr is the data header pointer. It is -352 bytes relative the DBF data header. However, rather than using absolute addressing to locate field descriptor data (needed for parsing), it's recommended that the DBF handle be obtained from the KHptr structure. Since no file handles are passed, you must read the xbLink handle value from the index file header. The xbLink handle is stored at KHptr+12. With this handle, you call the GET_DESCRIPTOR_XB routine to obtain field descriptor info for each field.

This translated key expression structure, and text expression location, is at:

```
// (This is an excerpt from the IX3 header format)
// Translated key expression as done by Parser during CreateIndex and Reindex.
// For each key part in KH.expression a 4-byte structure is used:

typedef struct _XLATEX {
 CHAR  ftype;   // field type (C,N,L, etc.),if bit7=1 and C then do UPPER key
 CHAR  length;  // bytes to use starting at offset (never > 64)
 SHORT offset;  // byte offset into data record that length bytes are to be used
} XLATEX;       // (note that offset does not count tag field byte)

ULONG xlateCount;           // 044 number of key fields (64/4=16 max fields)
XLATEX xlateExpression[16]; // 048 key construct info (16 dword's worth)
      :                     // 112-347   :
CHAR  expression[160];      // 348 key expression, user (0-Terminated)
```

xlateExpression is at offset +48 relative the IX3 index header. However, KHptr, passed to this routine, points to -384 relative the IX3 index header start. Therefore, to locate to xlateExpression, you must add 384 to 48. This means that xlateExpression[0].ftype is located at KHptr+432. The number of valid key components in xlateExpression is stored in xlateCount (at KHptr+428). The text key expression string, which you are to parse, is located at KHptr+732. This is identical to the expression passed during CREATE_INDEX_XB (and it is CREATE_INDEX_XB that calls this parser routine).

The calling convention for your custom expression parser function is APIENTRY, and the parameters are passed to your function on the stack (by Bullet). A sample prototype for a build-key function follows:

```
ULONG APIENTRY YourKeyExpressionParser(PVOID DHDptr,
                                       PVOID KHptr,
                                       PULONG keyLenPtr);
```

You are to parse the text key expression at KHptr+732 and store the key component XLATEX structure values to the XLATEX structure, one for each key component parsed. In addition, the key length (the sum of the XLATEX.length fields) is placed in the ULONG data pointed to by keyLenPtr. The keylength may not exceed 64 bytes. If DUPS_ALLOWED is flagged, add two to the sum of the XLATEX.length fields for the enumerator word.

Note: The key expression has been mapped to upper-case and 0-filled by the time this routine is called.

This is probably the most difficult part of customizing Bullet. However, the difficulty lies not with Bullet, but how you parse. The idea is simple — you are to generate a xlateCount value, and for each key component (*ie* non-contiguous, non-same-type run in the data record), an XLATEX variable describing the method to build that key component out of the data record (type, length, and starting offset) is stored. The text key expression is available in the index header, and the destination to write to is there, also. You do need to read the index header at KHptr+12 (ULONG) to obtain the DBF handle for this index file before you can parse the expression. This because you need to know about the record field names, types, and lengths before you can parse the key expression. The matter not covered here is that of parsing the expression, which is left to the programmer. Any lexical parser algorithm may be used, or you may even do no parsing at all, and simply hard-code values into the XLATEX structures.

If you've gotten this far, you may find the following data structures useful. The numbers at // nnn are offsets relative the SIP.herePtr and SDP.herePtr pointers. For example, at SIP.herePtr+352 is a ULONG of the number of key searches requested. These could be monitored in a separate thread.

Relative SIP.herePtr:

```
ULONG fType;            // 000 bit0=0 for index file, btree
ULONG flags;            // 004 bit0=1 is dirty
                        //     bit1=1 full lock (count stored in KH.lockCount)
                        //     bit2=1 shared lock (if bit1=1)
                        //     bit3-14 reserved (=0)
                        //     bit15=1 no coalesce on key delete
                        // 006 BYTE, progress of reindex (0,1-99)
                        // 007 BYTE, 0
PVOID morePtr;          // 008 ptr to additional header info, if ever needed
ULONG xbLink;           // 012 related XB data file handle
ULONG asMode;           // 016 access-sharing-cache mode of open
CHAR  filename[260];    // 020 filename at open (0T)
ULONG currKeyRecNo;     // 280 current rec number assigned to KH.currKey
CHAR  currKey[64];      // 284 current key value
ULONG rez0;             // 348 allow for 0-terminated string
ULONG searches;         // 352 keys searched for
ULONG seeks;            // 356 nodes seeked
ULONG hits;             // 360 seeks satisfied without disk access
ULONG keysDeleted;      // 364 keys deleted since last zeroed
ULONG keysStored;       // 368 keys added since last zeroed
ULONG nodesSplit;       // 372 splits needed on insert since last zeroed
ULONG nodesMadeAvail;   // 376 nodes made available from deleting keys
ULONG lockCount;        // 380 active full-lock count

// the IX3 index header follows at 384+
```

Relative SDP.here:

```
ULONG fType;            // 000 bit0=1 for DBF data file, XB
ULONG flags;            // 004 bit0=1 is dirty
                        //     bit1=1 full lock
                        //     bit2=1 shared lock (if bit1=1)
                        //     bit3-15 reserved (=0)
                        // 006 BYTE, progress of pack (0,1-99)
                        // 007 BYTE, 0
PVOID morePtr;          // 008 ptr to additional header info, if ever needed
ULONG noFields;         // 012 number of fields in this data file
ULONG asMode;           // 016 access-sharing-cache mode of open
CHAR  filename[260];    // 020 filename at open (0T)
ULONG lockCount;        // 280 only when dec'ed to 0 do full unlock
ULONG memoAvailBlock;   // 284 next available block (header is block 0)
ULONG memoUnk1;         // 288 not used
CHAR  memoFilename[8];  // 292 filename proper (first 8 of filename proper)
ULONG memoUnk2;         // 300 not used (apparently)
ULONG memoBlockSize;    // 304 block size, must be at least 24 to cover header!
ULONG memoHandle;       // 308 handle of open memo file
ULONG memoFlags;        // 312 bit0=1 is dirty
ULONG memoLastNo;       // 316 last accessed memo number (if not 0)
ULONG memoLastLink;     // 320 link data for last accessed memo
ULONG memoLastSize;     // 324 size of last accessed memo (in bytes, w/OH)
ULONG align32[6];       // 328 (align to even32)

// the DBF data header follows at +352
```

License Agreement

Before using BULLET you must agree to the following:

1. A BULLET license grants you the right to use the BULLET library code on a royalty-free basis according to the terms of this License Agreement.
2. You are not permitted to operate more than one copy of this software package at one time per license. For example, if you have ten programmers that have *access* to the BULLET package at the same time, you are required to have ten BULLET licenses.
3. There is no restriction on the number of users you may support, and no restriction on the number of different end-user programs you may distribute that use BULLET. You may allow any number of simultaneous users to use your end-user program.
4. The dynamic link library, BULLET*.DLL, may be distributed with your end-user program. No other BULLET product may be distributed without permission (for example, you may not distribute Bullet's import library, BULLET*i.LIB).
5. You are not permitted to distribute non-executable code containing BULLET code. This means that may not redistribute BULLET with your program if your program can be used by other programmers to develop executable code. BULLET must be part of an end-user product only. This means that you cannot provide an overlay or other such external code containing BULLET code if that code is to be used as a programming library for other programmers, from which the other programmers can create programs. If you require distributing a non-end-user package containing BULLET, you must obtain written permission from the author. This limitation pertains to the distribution of BULLET library code. You may, however, develop and distribute your programmer package (i.e., non-end-user) as you wish, but you may not distribute BULLET library code with that package without written permission. For example, you may develop class libraries that use the BULLET library code, and distribute those tools that you have written, but you may not include BULLET library code, or BULLET activation methods, in that package. The programmer using your package would need a BULLET license to make use of your package.
6. The static link library, BULLET*.LIB, may *not* be distributed except in executable form as a component in your executable program (EXE). BULLET*.LIB may not be placed into a DLL. BULLET*.LIB, if part of your license/Option, may only be linked directly to your end-user program. BULLET*.LIB is available with Option C and E licenses only.
7. Shareware use is limited to 28 days, and for the sole purpose of evaluating the software. The BULLET library code may not be distributed in any form without a registered BULLET license. A BULLET license is obtained only with purchase of a BULLET package, purchased from an authorized BULLET distributor (see Order Information in the online manual, or the !ORDER.FRM and !ORDER.CC files).
8. A BULLET license is specific to the option level purchased. License holders with a lower Option may *not* use any higher level Option code. For example, if you find another product using the Option C DLL, and you have an Option A license, you are not permitted to use the Option C DLL in your development, nor may you distribute any code that is not part of your Option level.
9. Your end-user program using the BULLET DLL is required to be a copyrighted work, and must contain a valid copyright notice in the form, 'Program-name Copyright ©Year Your-Name', or similar. No notice of BULLET's copyright need be further specified in your program (in other words, you don't need to mention BULLET, but you may if you wish). This applies only if you distribute the BULLET DLL with your product. Programs that are linked using the BULLET*.LIB static link library need not display a copyright notice. In other words, if you must distribute to the Public Domain, where no copyright is desired for your program, you must link using the static link library, and not the DLL.
10. BULLET is owned by the author, Cornel Huth, and is protected by United States copyright laws and international treaty provisions. You are not permitted to make copies of this software except for archival purposes.
11. 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.
12. You may not reverse engineer, decompile, or disassemble this software if the intent or result is to alter the software.
13. There are no expressed or implied warranties with this software.
14. All liabilities in the use of this software rest with the user.
15. 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. The software is owned by Cornel Huth/6402 Ingram Rd/San Antonio Texas 78238/USA. This agreement is governed by the laws of the Great State of Texas, the United States of America, and all other countries of Earth.

Any questions concerning this License Agreement should be directed to me at any of the addresses listed under Product Support.

Note: Failure to comply with any part of this License Agreement immediately terminates any and all licenses that you may have to use BULLET.

Installation and Product Support

Installation instructions are located in the README text file included with your package.

Technical support in the use of Bullet is available for licensed users at the 40th Floor BBS or by way of the Internet.

```
40th Floor BBS: +1(210)684-8065  N-8-1

Internet e-mail:  support@40th.com
            www:  http://www.40th.com
```

The latest in-version (2.x) release of BULLET/2 is available for download from the BBS by registered users, or $5 by postal mail. Re-createable bugs are fixed immediately. Report a program bug and get the fix shipped free. Past bugs are listed at the BBS and the web site. Contact support if you have any questions or requests.

For general information send e-mail to:

```
            info@40th.com
```

Bug Report Form

When requesting support for possible bug(s) use the following as a guideline:

1. Include a complete problem description.
2. Include sample source of the problem, if necessary (small is best).
3. Include necessary data files, include files, etc.
4. Include step-by-step procedure to follow in order to recreate the problem.

Once done, send it to support by way of the BBS, e-mail, or postal mail to addresses listed in Product Support.

Index

Bottom Notes