*From the perspective of a programmer, lq-text is made up of several libraries: 1. liblqtext is the main text retrieval library; 2. liblqutil defines a number of useful general routines, mostly for file and string manuipulation, and 3. liblqerror deals with error handling.*

*You'll probably end up using a mixture of these libraries in practice. If you link against liblqtext you will also need to link against the other libraries; if you link against liblqutil you will need to link against liblqerror, but you can use liblqutil without liblqtext. In the same way, liblqerror stands alone and can be used without any of the other libraries.*

*You will therefore need to link with the options '-llqtext' '-llqutil' '-llqerror' in that order. You may also need to use '-L' to specify the directory containing the libraries, depending on your installation, and '-I' to name the directory with the include files. The default places are '/usr/local/lib/lqtext' and '/usr/local/lib/lqtext/include' for these, respectively.*

*The functions in each library all have a common prefix, starting with* LQ, *as follows: 1.* LQT *is used by text retrieval routines; 2.* LQU *is used by liblqutil routines; 3.* LQM *is used for a number of memory-related routines; 4.* LQE *is used for error handling routines.*

*Since efficiency is a major part of text retrieval, some of the routines documented here are actually implemented as macros. It is deliberately unspecified as to which these are, so that you can't rely on the implementation. You must therefore avoid having side-effects in parameters to function calls, as any parameter may be evaluated multiple times.*

*Some of the routines have an '_' after their prefix, and some have a 'p' instead; the 'p' is a reminder that the routine concerned is private to the lqtext libraries, and shold not normally be used in client software.*

*To make them easier to copy and use in other programs, many of the example clients shipped with lq-text have function names that begin with* LQC_ *(the* C *stands for* Client*), so that they are unlikely to conflict with anything you're already using. The sample clients are located in the lq-text/src/lqtext directory; the samples in this manual are in the lq-text/doc/samples directory.*

¶ *Functions in this category are related to manipulating an lq-text database as a whole.*

*Usually when you are working with lq-text, you will first call* LQT_InitFromArgv (*in the* Database/Defaults *category*), *and then pass the return value from that as an argument to* LQT_OpenDatabase; *before exiting, you should normally call* LQT_CloseDatabase.

---

Database/Database
../liblqtext/close.c

```
API void
LQT_AddActionOnClose(db, Description, Action, Flags)
  t_LQTEXT_Database *db;
  char *Description;
  int (* Action)(
    t_LQTEXT_Database *
  );
  unsigned int Flags;
```

The given Action function will be called whenever LQT_CloseDatabase or LQT_SyncDatabase is called. ¶ The string Description is used in trace and debugging messages, and also in error messages; it should be a human-readable description of the action that the function is performing, or it could be an ASCII string containing the name of the function. The Description string is not copied; a pointer to it is retained. Therefore, it is an error to free it after calling LQT_-AddActionOnClose. ¶ The given Flags argument may be any combination of LQT_ON_-SYNC and LQT_ON_CLOSE using bitwise or. If the LQT_ON_SYNC flag is given, the given Action is called by LQT_SyncDatabase; if LQT_ON_CLOSE is given, the given Action is called by LQT_CloseDatabase. It is unusual to have an action for the Sync case and not for the Close case, but it is not forbidden. ¶ You can register any number of functions in this way. The most recently registered function is called first, and so on. ¶ LQT_-OpenDatabase uses this function to register the following functions, in order, so that LQT_FlushBlockCache is called last: 1. LQT_FlushBlockCache (Write out cached low-level data blocks); 2. LQT_WriteCurrentMaxWID (Write out largest allocated WID); 3. LQTpFlushWIDCache (Write out cached WID index blocks); 4. LQTp_-FlushLastBlockCache (Write out cached lastblock data); 5. LQT_-

SyncAndCloseAllKeyValueDatabases (Flush and close all open Dynamic Hashing (ndbm) key-value databases) ¶ You can see these called by running a client with the Debug trace flag set (e.g. lqwordlist -t Debug). ❦ Notes: The list of functions registered may change between revisions of lq-text, and is given here for illustrative purposes. ❦ See Also: LQT_OpenDatabase‡ (opposite); LQT_CloseDatabase‡ (below); LQT_OpenKeyValueDatabase‡ (p. 38).

---

Database/Database
../liblqtext/lqrverno.c

API void
**LQT_CheckDatabaseVersion**(db)
  t_LQTEXT_Database *db;

Checks that the current database is compatible with this version of the library. Some versions of liblqtext may have a backwards compatibility mode, which this function will enable. This routine is called automatically whenever an lq-text database is opened. ❦ Notes: The liblqtext library is capable of maintaining backward compatibility with earlier versions; for example, Release 1.13 could read a database created with Release 1.12; this feature is not presently included, however. In practice, it's almost always possible to index the data again rather than using backwards compatibility modes, and performance is usually then better. ❦ Errors: Fatal error if the database is incompatible with the current version of the lqtext library.

---

Database/Database
../liblqtext/close.c

API int
**LQT_CloseDatabase**(theDatabase)
  t_LQTEXT_Database *theDatabase;

Closes the current lq-text database. Any actions that have been registered with LQT_AddActionOnClose are performed, including the ones that liblqtext has registered. It is not necessary to call LQT_SyncDatabase before closing a database, as LQT_CloseDatabase does this. All pending data is flushed, and all file descriptors that have been opened by liblqtext functions are closed. Currently, not all allocated memory is freed, but any such memory is not lost, because it will be reused on a subsequent call to LQT_OpenDatabase. ❦ Returns: zero. ❦ See Also: LQT_AddActionOnClose‡ (previous page); LQT_OpenDatabase‡ (opposite); LQT_SyncDatabase‡ (opposite).

---

Database/Database,
Database/Files
../liblqtext/smalldb.c

API int
**LQT_CurrentlyHaveWriteAccess**(db)
  t_LQTEXT_Database *db;

Returns non-zero if and only if the given database is open with write access. ❦ Notes: Write access may on some systems be exclusive, so that no other process can open the database, neither for reading nor for writing. You should not rely on this, however. ❦ See Also: LQT_ObtainWriteAccess‡ (opposite); LQT_OpenDatabase‡ (opposite).

**LQT_AddActionOnClose** Database/Database, p. 3
**LQT_CloseDatabase** Database/Database, p. 4
**LQT_ObtainWriteAccess** Database/Database, Database/Files, p. 5
**LQT_OpenDatabase** Database/Database, p. 5
**LQT_OpenKeyValueDatabase** Database/Dynamic Hashing, Database/Files, p. 38
**LQT_SyncDatabase** Database/Database, p. 5

API int
**LQT_ObtainReadOnlyAccess**(db)
  t_LQTEXT_Database *db;

Obtains read-only access to the current database. This is called automatically by LQT_OpenDatabase if appropriate. If the database was previously open for writing, it should be closed first with LQT_CloseDatabase or LQT_SyncDatabase. ❦ Returns: 1. zero on success 2. −1 on failure or error ❦ Errors: A corrupt database may cause a fatal or E_BUG error. ❦ See Also: LQT_OpenDatabase‡ (below); LQT_ObtainWriteAccess‡ (below); LQT_OpenKeyValueDatabase‡ (p. 38); LQT_CloseDatabase‡ (opposite).

Database/Database, Database/Files ../liblqtext/smalldb.c

---

API int
**LQT_ObtainWriteAccess**(db)
  t_LQTEXT_Database *db;

Grants write access to the current database. This is called automatically by LQT_OpenDatabase if appropriate. ❦ Returns: 1. zero on success 2. −1 on error or failure ❦ Notes: Write access may on some systems be exclusive, so that no other process can open the database, neither for reading nor for writing. You should not rely on this, however; on some systems, multiple clients may succeed in writing, and will corrupt the database. ❦ Errors: A corrupt database may cause a fatal or E_BUG error. ❦ See Also: LQT_ObtainReadOnlyAccess‡ (above); LQT_OpenDatabase‡ (below).

Database/Database, Database/Files ../liblqtext/smalldb.c

---

API t_LQTEXT_Database *
**LQT_OpenDatabase**(Options, flags, modes)
  t_lqdbOptions *Options;
  int flags;
  int modes;

Opens the lq-text database referred to in the given Options object; flags and modes are as for open(2), although in all cases the lq-text directory must already exist. ¶ The only valid Options object at the moment is the value returned by LQT_InitFromArgv, which can only be called once during the lifetime of a process. ❦ Notes: Since you can currently only have a single database open in any given program, there is not yet a need for a way to open a specific database; this will change in the next release. ❦ Returns: A pointer to an opaque object describing the database. The pointer is suitable for use with LQT_CloseDatabase. ❦ See Also: LQT_InitFromArgv‡ (p. 7); LQT_CloseDatabase‡ (opposite).

Database/Database ../liblqtext/open.c

---

API int
**LQT_SyncDatabase**(theDatabase)
  t_LQTEXT_Database *theDatabase;

Syncs the current lq-text database; that is, writes any pending data blocks to disk, and closes and deletes any temporary files. You could think of this function as closing the database and then opening it again, except that it doesn't actually do that. ¶ Any actions that have been registered with LQT_AddActionOnClose with the lqt_on_sync flag are performed, including the ones

Database/Database ../liblqtext/close.c

that liblqtext has registered internally. ¶ It is not necessary to call LQT_SyncDatabase before closing a database, as LQT_CloseDatabase does this. ❦ Returns: zero. ❦ See Also: LQT_AddActionOnClose‡ (p. 3); LQT_OpenDatabase‡ (previous page); LQT_CloseDatabase‡ (p. 4).

*This category relates to user preferences, which may be found in the database configuration file* README, *in the environment or on the command-line on operating systems where those make sense, or in a per-user configuration file.*

*Currently, most of the preferences and configuration code exists only as a set of place-holders.*

*A program should do the following on startup: 1. Set the global variable progname to a useful value; 2. Call* LQT_InitFromArgv *to obtain an* Options *object; this may result in argv being changed. 3. Handle any program-specific command-line options; 4. Call* LQT_OpenDatabase *with the* Options *object obtained earlier, and also with* O_RDONLY *or* O_RDWR *as appropriate.*

*Finally, on exit, the program should call* LQT_CloseDatabase*; this is optional if you only used read-only access to the database.*

---

```
API void *
LQT_GetOption(Options, Name)
  t_lqdbOptions *Options;
  char *Name;
```

Database/Defaults
../liblqtext/defaults.c

This function returns the value of a configuration option. The options at present include: 1. 'directory', which is the name of the directory containing the lq-text database; 2. 'stop list', which is the name of a file containing words that are not indexed; 3. 'file search path', which is a colon-separated list of directories that are searched for documents during indexing and retrieval, and 4. 'phrase match level', which determined how precisely phrases are matched. ❦ Returns: A pointer to the actual value; do not free this value. ❦ See Also: LQT_OpenDatabase‡ (p. 5).

---

```
API t_lqdbOptions *
LQT_InitFromArgv(argc, argv)
  int argc;
  char **argv;
```

Database/Defaults
../liblqtext/defaults.c

This function is called to Initialise the lq-text libraries. It sets the global variable 'progname' from argv[0], but does not remove any leading directories; if you want just the command name to appear in error messages and other output, you should set progname in main() before calling LQT_InitFromArgv. ¶ After setting progname, LQT_InitFromArgv handles any lq-text command-line options. Currently, each option is turned into either -z if it does not take an argument, or -Z if it take an argument. As a result, you should ignore -z and -Z options if they appear, together with the argument to -Z, and you should not give your program a -z or -Z option. This behaviour will change completely in a future release of lq-text, when improved command-line argument handling is introduced. ¶ The command line options currently understood include: 1. -d *dir*, to specify a database directory 2. -m *p*|*h*|*a*, to specify whether to match phrases precisely, heuristically, or approximately; 3. -t *flags*, to turn on tracing; the given flags should be a string of debugging flag names separated by the vertical bar (|). An example would be -t Trace|Debug, but you will usually need to quote the argument to protect it from the shell. The value List will print a list of available values. ❦ Restrictions: Must be called before any other liblqtext functions. ❦ Returns: A pointer to an object used to represent options; this object should be passed to LQT_OpenDatabase(). ❦ See Also: LQT_OpenDatabase‡ (p. 5).

---

Database/Defaults, Database/Words, Language/Stemming ../liblqtext/ctype.c

LIBRARY void
**LQTp_InitialiseCharacterTypes**(db)
  t_LQTEXT_Database *db;

Initialises the tables used to determine whether a given character is part of a word or not. ¶ This function is called automatically by LQT_OpenDatabase(). ❦ Returns: zero on success.

---

Database/Defaults ../liblqtext/defaults.c

API void
**LQT_PrintDefaultUsage**(Options)
  t_lqdbOptions *Options;

Prints to stderr a usage message that describes command-line options specific to (and interpreted by) liblqtext. You should call this if an unknown command-line option was found, other than -z or -Z. ❦ Notes: This routine will change in the next release, with an entirely new argument processing mechanism. ❦ See Also: LQT_InitFromArgv‡ (previous page).

*Functions in this category are used to get information back from the database. This is what it's all about: the functions this category are the main rationale for the existance of the database package.*

---

API char *
**LQT_FIDToDocumentTitle**(db, FID, Name)
  t_LQTEXT_Database *db;
  t_FID FID;
  char *Name;

Database/Retrieval, Database/Documents ../liblqtext/gettitle.c

Returns a document title (from the database 'titles' file) for a given FID. A binary search is used to locate a line in the titles file which starts with the given FID, as a decimal ASCII number, followed by a tab; the remainder of that line up to a newline or EOF is returnd. The second (Name) argument is only used on error. ❦ Returns: 1. the title on success, in a static buffer 2. The given Name pointer on error. ❦ Errors: Warns if the title file can't be opened. ❦ Notes: The 'lqkwic' client uses this function to expand ${Title}.

---

API t_FileInfo *
**LQT_FIDToFileInfo**(db, FID)
  t_LQTEXT_Database *db;
  t_FID FID;

Database/Retrieval, Database/Documents ../liblqtext/fileinfo.c

Returns the in-memory t_FileInfo struct associated with a given FID, reading the information from the database as necessary. The returned value, if non-zero, is created with malloc; it is the caller's responsibility to free the storage. ❦ See Also: LQT_NameToFID‡ (p. 14); LQT_DestroyFileInfo‡ (p. 35). ❦ Returns: 1. the t_FileInfo * on success;

**LQT_DestroyFileInfo** Database/Documents, Memory, p. 35 **LQT_NameToFID** Database/Retrieval, Database/Documents, p. 14

2. NULL on error. ❦ Errors: Warns if the database can't be opened. ❦ See Also: LQT_NameToFID‡ (p. 14); LQT_DestroyFileInfo‡ (p. 35).

---

Database/Retrieval, Database/Words ../liblqtext/wordinfo.c

```
API t_WID
LQT_FindFirstWIDMatchingPattern(
  db,
  Pattern,
  PatternLength,
  PrefixLength,
  Matcher,
  Argument )
  t_LQTEXT_Database *db;
  unsigned char *Pattern;
  int PatternLength;
  int PrefixLength;
  int (* Matcher)(
    /* prefix with 'the' in order to avoid old gcc bug */
    t_LQTEXT_Database *thedb,
    unsigned char *theString,
    int theStringLength,
    unsigned char *thePattern,
    int thePatternLength,
    int thePrefixLength,
    unsigned char *theArgument
  );
  unsigned char *Argument;
```

Returns the lowest WID whose word matches the given Pattern. ¶ The Pattern need not be NUL-terminated; the given PatternLength argument is used to find the end of the Pattern. ¶ The given PrefixLength argument must specify the number of leading characters, if any, in the given Pattern that form a constant prefix. If there are no such characters, matching is likely to be several orders of magnitude slower, as LQT_FindFirstWIDMatchingPattern will have to try every word in the database vocabulary, one at a time, until it finds one that matches. ¶ The given Matcher argument must be a pointer to a function that will try to match the string to the given pattern, and that will return zero only on a match. The constant LQT_WIDMATCH_FAILED is available in <liblqtext.h> to be returned by the given Matcher function, indicating that LQT_FindFirstWIDMatchingPattern should fail and return zero immediately. This might be used if the given Matcher function is called with a string lexically greater than the largest that could ever match it, or after reporting an error. ¶ The given Argument is passed on to the Matcher function, for the convenience of the caller. ❦ Returns: The WID on success, and zero on failure. ❦ Errors: Warns if a database format error is detected. ❦ See Also: LQT_WordToWID‡ (p. 16).

**LQT_DestroyFileInfo** Database/Documents, Memory, p. 35
**LQT_NameToFID** Database/Retrieval, Database/Documents, p. 14
**LQT_WordToWID** Database/Retrieval, Database/Words, p. 16

API t_WID
**LQT_FindFirstWIDMatchingPrefix**(db, Prefix, PrefixLength)

Database/Retrieval, Database/Words ../liblqtext/wordinfo.c

```
t_LQTEXT_Database *db;
char *Prefix;
int PrefixLength;
```

Returns the lowest WID whose word matches the given Prefix. ¶ The Prefix need not be nul-terminated; the given PrefixLength argument is used to find the end of the Prefix. ❦ Returns: The WID on success, and zero on failure. ❦ Errors: Warns if a database format error is detected. ❦ See Also: LQT_WordToWID‡ (p. 16).

API t_OffsetPair *
**LQT_FindMatchEnds**(db, Buffer, Length, StartBlock, BIF, WIB, NumberOfWords)

Database/Retrieval, Database/Documents ../liblqtext/matchend.c

```
t_LQTEXT_Database *db;
char *Buffer;
unsigned int Length;
char *StartBlock;
unsigned long BIF;
unsigned long WIB;
int NumberOfWords;
```

Returns pointers to the start and end of the matched text in the given buffer. LQT_FindMatchEnds must be called with at least one block of data (FILEBLOCKSIZE in `<globals.h>`, usually 64 bytes) either side of the block containing the match. Providing more blocks before the matched block is more likely to result in a correct return value, as there are some special cases involving words spanning block boundaries that are best dealt with by looking a block further back until a block boundary is found that has a space to one side of it, and LQT_FindMatchEnds does this. ¶ The Buffer argument is the text from the file, with StartBlock being a pointer to the first character in the block containing the match. The BIF and WIB arguments are the Block In File and Word In Block fields from the match, and the NumberOfWords argument determines the number of words in the match, for setting the match end pointer. ❦ Returns: 1. a t_OffsetPair on success, containing pointers to the first matched character and the last matched character. 2. zero if the match wasn't found ❦ See Also: LQT_ReadWordFromStringPointer‡ (p. 15).

API t_WID
**LQT_FindNextWIDMatchingPattern**(

Database/Retrieval, Database/Words ../liblqtext/wordinfo.c

```
  db,
  WID,
  Pattern,
  PatternLength,
  PrefixLength,
  Matcher,
  Argument )
  t_LQTEXT_Database *db;
  t_WID WID;
  unsigned char *Pattern;
  int PatternLength;
  int PrefixLength;
  int (* Matcher)(
    t_LQTEXT_Database *thedb,
    unsigned char *theString,
    int theStringLength,
    unsigned char *thePattern,
    int thePatternLength,
    int thePrefixLength,
    unsigned char *theArgument
  );
  unsigned char *Argument;
```

Returns the lowest WID whose word matches the given pattern, and that is greater than the given WID argument. The pattern is a string, which must be an all-lower-case prefix. The given wildcard character must be either * or ?, to indicate zero or more following characters or exactly one following character, respectively. ¶ The Prefix need not be nul-terminated; the given PrefixLength argument is used to find the end of the prefix. ❦ Returns: The WID on success, and zero on failure. ❦ Errors: Warns if a database format error is detected. ❦ See Also: LQT_FindFirstWIDMatchingPattern‡ (p. 10).

---

Database/Retrieval, Database/Words ../liblqtext/wordinfo.c

API t_WID
**LQT_FindNextWIDMatchingWildCard**(db, WID, Prefix, PrefixLength)

```
  t_LQTEXT_Database *db;
  t_WID WID;
  char *Prefix;
  int PrefixLength;
```

Returns the lowest WID whose word matches the given pattern, and that is greater than the given WID argument. The pattern is a string, which must be an all-lower-case prefix. The given wildcard character must be either * or ?, to indicate zero or more following characters or exactly one following character, respectively. ¶ The Prefix need not be nul-terminated; the given PrefixLength argument is used to find the end of the prefix. ❦ Returns: The WID on success, and zero on failure. ❦ Errors: Warns if a database format error is detected. ❦ See Also: LQT_FindFirstWIDMatchingWildCard (undocumented);

---

Database/Retrieval, Database/Physical ../liblqtext/smalldb.c

API void
**LQT_GetFileModes**(db, Flagsp, Modesp)

```
  t_LQTEXT_Database *db;
  int *Flagsp;
  int *Modesp;
```

Returns the current file modes, as determined by LQT_ObtainReadOnlyAccess or LQT_ObtainWriteAccess, in Flagsp and Modesp. The returned values are suitable for passing to open(2). ❦ Errors: Passing null pointers causes a fatal (E_BUG) error. ❦ See Also: LQT_OpenDatabase‡ (p. 5); LQT_ObtainWriteAccess‡ (p. 5); LQU_Eopen‡ (p. 72).

**LQT_FindFirstWIDMatchingPattern** Database/Retrieval, Database/Words, p. 10
**LQT_ObtainWriteAccess** Database/Database, Database/Files, p. 5
**LQT_OpenDatabase** Database/Database, p. 5
**LQU_Eopen** Utilities/Files, p. 72

API t_WordPlace *
**LQT_GetWordPlaces**(db, WID, Block, BlockLength, NextOffset, NumberExpected)

```
  t_LQTEXT_Database *db;
  t_WID WID;
  unsigned char *Block;
  unsigned int BlockLength;
  unsigned long NextOffset;
  unsigned long *NumberExpected;
```

Database/Retrieval, Database/Physical ../liblqtext/rpblock.c

Reads all the places for a given word into memory, and returns a freshly malloc'd array of t_WordPlaces. It is the caller's responsibility to free the resulting array. ¶ The arguments are as for LQT_GetWordPlacesWhere. ❦ See Also: LQT_GetWordPlacesWhere[‡] (below); LQT_MakeMatchesWhere[‡] (p. 17).

---

API t_WordPlace *
**LQT_GetWordPlacesWhere**(

```
  db,
  WID, Block, BlockLength,
  NextOffset,
  NumberExpected,
  AcceptFunc )
  t_LQTEXT_Database *db;
  t_WID WID;
  unsigned char *Block;
  unsigned int BlockLength;
  unsigned long NextOffset;
  unsigned long *NumberExpected;
  int (* AcceptFunc)(
    t_LQTEXT_Database *,
    t_WID,
    t_WordPlace *
  );
```

Database/Retrieval, Database/Physical ../liblqtext/rpblock.c

Used to read the matches from disk for the given WID. ¶ A WordPlace describes a single occurrence of a word. Hence, if you call this function with the WID of 'the', you'll get back an array large enough to hold every occurrence of 'the' in the entire database. The AcceptFunc argument is a function that is called before each match is inserted into the array; it can return either zero or one. If it returns zero, the match is not inserted into the array; this can save memory, and also allows you to process the matches as they are read from disk, instead of waiting for them all before doing anything with them. ¶ The given Block argument is a pointer to an in-memory buffer holding the first few bytes of data; usually this comes from the 'widindex' fixed record length file. ❦ Notes: This function is very low-level; normally, you should use LQT_MakeMatches or LQT_MakeMatchesWhere instead. ❦ See Also: LQT_GetWordPlaces[‡] (above); LQT_GetpblockWhere[‡] (overleaf); LQT_StringToPhrase[‡] (p. 22); LQT_MakeMatchesWhere[‡] (p. 17).

---

API t_pblock *
**LQT_Getpblock**(db, WordInfo)

```
  t_LQTEXT_Database *db;
  t_WordInfo *WordInfo;
```

Database/Retrieval, Database/Physical ../liblqtext/rpblock.c

Returns a freshly malloc'd t_pblock containing all of the WordPlaces for a given WordInfo; one for each occurrence of that word in the database. ❦ Returns: 1. the number of words added on success; 2. −1 if the file couldn't be opened. ❦ Errors: Warns if the file can't be opened. ❦ See Also: LQT_GetpblockWhere[‡] (overleaf).

**LQT_GetWordPlaces** Database/Retrieval, Database/Physical, p. 13
**LQT_GetWordPlacesWhere** Database/Retrieval, Database/Physical, p. 13
**LQT_GetpblockWhere** Database/Retrieval, Database/Update, Database/Physical, p. 14
**LQT_MakeMatchesWhere** Retrieval/Matching, Retrieval/Phrases, p. 17
**LQT_StringToPhrase** Retrieval/Phrases, p. 22

Database/Retrieval,
Database/Update,
Database/Physical
../liblqtext/rpblock.c

```
API t_pblock *
LQT_GetpblockWhere(db, WordInfo, AcceptFunc)
  t_LQTEXT_Database *db;
  t_WordInfo *WordInfo;
  int (* AcceptFunc)(
    t_LQTEXT_Database *,
    t_WID,
    t_WordPlace *
  );
```

Look up a word in the database... and return a liſt of all the WordPlaces where it's found. The AcceptFunc is called for each place as it is read off the disk, with the given db, the WID and the new WordPlace as arguments. If the AcceptFunc returns a positive value, the WordPlace is accepted; otherwise, it is not included in the returned t_pblock. Note that it is possible to end up with a pblock with no WordPlaces at all if the AcceptFunc never returns a positive value. An AcceptFunc of NULL is considered to return 1 in every case. ❦ Returns: a freshly malloc'd t_pblock containing all of the WordPlaces from the disk that the AcceptFunc accepted, and with NumberOfWordPlaces set to the number of such places. ❦ Notes: Normally you would use LQT_MakeMatches inſtead of this function. This function is used internally, and also by lq-text clients that update the database efficiently. ❦ Errors: Database format errors are nearly always fatal. ❦ See Also: LQT_MakeMatches‡ (p. 17).

Database/Retrieval,
Database/Documents
../liblqtext/fileinfo.c

```
API t_FID
LQT_NameToFID(db, Name)
  t_LQTEXT_Database *db;
  char *Name;
```

Returns the FID associated with a given file name ❦ Returns: 1. the FID on success 2. zero on failure ❦ See Also: LQT_FIDToFileInfo‡ (p. 9); LQT_GetMaxOrAllocateFID‡ (p. 36). ❦ Errors: Warns if the database can't be opened. If the filename is not matched in the database, no warning is given, but zero is returned.

Database/Retrieval,
Database/Documents
../liblqtext/readword.c

```
API t_WordInfo *
LQT_ReadWordFromFileInfo(db, FileInfo, Flags)
  t_LQTEXT_Database *db;
  t_FileInfo *FileInfo;
  unsigned int Flags;
```

The same as LQT_ReadWordFromStringPointer, but uses a FILE * that the caller has created in the given t_FileInfo ſtructure. ❦ Notes: See LQC_MakeInput in the lqaddfile client for one way to create a FileInfo; that routine will move into the API in a future release, but probably with slight changes to its interface. ❦ See Also: LQT_ReadWordFromStringPointer‡ (opposite).

**LQT_FIDToFileInfo** Database/Retrieval, Database/Documents, p. 9
**LQT_GetMaxOrAllocateFID** Database/Documents, p. 36
**LQT_MakeMatches** Retrieval/Matching, Retrieval/Phrases, p. 17
**LQT_ReadWordFromStringPointer** Database/Retrieval, Database/Documents, p. 15

API t_WordInfo *
**LQT_ReadWordFromStringPointer**(db, Stringpp, Startp, Endp, Flags)
  t_LQTEXT_Database *db;
  char **Stringpp;
  char **Startp;
  CONST char *Endp;
  unsigned int Flags;

Database/Retrieval, Database/Documents ../liblqtext/readword.c

Returns the next natural-language word from the given NUL-terminated ſtring. ¶ The definition of a word for the purpose of this routine is determined partly by the definitions for LQT_StartsWord, LQT_OnlyWithinWord and LQT_EndsWord in the header file `<wordrules.h>`, and partly on the configuration file in the database directory, where indexnumbers, minwordlength and maxwordlength may be set. ¶ If the arguments are all null, the effect is to reset the routine ready to ſtart a new ſtring, and no useful value is returned in that case. ¶ The given Flags argument may either be zero or any combination of LQT_READWORD_IGNORE_COMMON and LQT_READWORD_WILDCARDS, or'd together. ¶ Characters are read from the ſtring, incrementing *Stringpp as each byte is processed, until a recognised word is found. If the LQT_READWORD_IGNORE_-COMMON flag was set in Flags, LQT_ReadWordFromStringPointer continues until either a word is found that has not been regiſtered as being too common to index, or the end of the ſtring is reached. ¶ If Startp is not a NULL pointer, *Startp is set to point to the firſt character in the word that has been found in the given Stringpp (not to the malloc'd copy in the result). ¶ If Endp is a NULL pointer, the ſtring is considered to be terminated by the firſt zero byte reached; otherwise, Endp muſt point to the firſt character not in the ſtring; normally, Endp would be set to point to the terminating NUL byte. ¶ If the LQT_-READWORD_WILDCARDS flag is set, the 'Wild Card' characters * and ? are allowed within words. Such characters do not count as punctuation for the returned WordInfo flags. ❦ Returns: the next WordInfo on success, or zero if there are no more words to read in the ſtring. ❦ Notes: All client programs and library routines which parse words use this routine or the companion LQT_ReadWordFromFileInfo routine. This is very important, because lq-text relies on word counts within each block of text to be the same on retrieval as they were on indexing, and if different routines parsed the data each time there would be a chance of discrepancies. ❦ Bugs: The interface to this routine is somewhat ugly, and may be changed in the next release with the addition of a Reset routine and a block offset counter.

API char *
**LQT_WIDToWord**(db, WID)
  t_LQTEXT_Database *db;
  t_WID WID;

Database/Retrieval, Database/Words ../liblqtext/wordinfo.c

Returns the word corresponding to a given WID. ❦ Returns: 1. the word on success 2. zero on failure, or if the wordliſt database parameter was set to off when the word was laſt written to the database ❦ Notes: LQT_WIDToWord may be inefficient or unavailable if the wordliſt parameter in the database config file is set to off. See the lqwordliſt program for alternate ways of obtaining access to the index vocabulary.

---

Database/Retrieval, Database/Words ../liblqtext/wordinfo.c

API t_WordInfo *
**LQT_WIDToWordInfo**(db, WID)
  t_LQTEXT_Database *db;
  t_WID WID;

Returns the in-memory WordInfo ſtructure for a given WID. ❦ Returns: 1. t_WordInfo * on success; 2. NULL on failure, or if th given WID argument was zero. ❦ Errors: Warns if a database format error is detected. ❦ See Also: LQT_WordToWID‡ (below).

---

Database/Retrieval, Database/Words ../liblqtext/wordinfo.c

API t_WID
**LQT_WordToWID**(db, Word, Length)
  t_LQTEXT_Database *db;
  char *Word;
  unsigned int Length;

Returns the WID for a given Word. It is not necessary that the word be NUL terminated. The Length argument is the number of bytes in the Word, not including any trailing NUL byte ❦ Returns: 1. the WID on success 2. 0 on failure ❦ See Also: LQT_WIDToWordInfo‡ (above). ❦ Errors: Fatal error if the database can't be opened.

## *Retrieval/Matching*

*This section describes routines used for matching words and phrases, and for fetching the results.*

---

API t_PhraseElement *
**LQT_AllPhrasesOfLengthNOrMore**(db, N, theQuery, Countp)
  t_LQTEXT_Database *db;
  int N;
  char *theQuery;
  long *Countp;

Retrieval/Matching,
Retrieval/Phrases
../liblqtext/phrall.c

Finds all sequences of N or more words which occur in the data. For example, given the phrase 'the barefooted boy was very slender', and supposing 'the' to be the only word for which LQT_WordIsInStopList returns true, LQT_AllPhrasesOfLengthNOrMore might find 'barefooted boy' and 'boy was very' and 'very slender' as sub-phrases that occur; if the entire phrase occurs, it will be returned. ¶ If a phrase of M words matches, all phrases of lengths from N to M inclusive will also be returned. ¶ It is the caller's responsibility to deallocate the returned array and its elements. ❦ Returns: an array of t_PhraseElement structures, and the number of distinct phrases found in *Countp. ❦ Notes: This function is experimental. It has not been optimised, and is currently unusable for long phrases as a result.

---

API long
**LQT_MakeMatches**(thedb, Phrase)
  t_LQTEXT_Database *thedb;
  t_Phrase *Phrase;

Retrieval/Matching,
Retrieval/Phrases
../liblqtext/phrase.c

This is equivalent to LQT_MakeMatchesWhere with a null AcceptFunction, and is provided for convenience. ❦ See Also: LQT_StringToPhrase‡ (p. 22); LQT_MakeMatchesWhere‡ (below).

---

API long
**LQT_MakeMatchesWhere**(db, Phrase, AcceptFunction)

Retrieval/Matching,
Retrieval/Phrases
../liblqtext/phrase.c

**LQT_MakeMatchesWhere** Retrieval/Matching, Retrieval/Phrases, p. 17
**LQT_StringToPhrase** Retrieval/Phrases, p. 22

```
t_LQTEXT_Database *db;
t_Phrase *Phrase;
int (*AcceptFunction)(
  t_LQTEXT_Database *,
  t_Phrase *,
  t_Match *
);
```

Matches the given phrase, and returns the number of successful matches. The given AcceptFunction is called for each match; it must return one of the following flags as defined in `<phrase.h>`: either LQMATCH_ACCEPT, which adds the match to the result, or LQMATCH_REJECT, which does not add the match to the result. In addition, either of these flags may be combined (using bitwise or) with LQMATCH_QUIT, in which case LQT_-MakeMatchesWhere will return the result collected so far and abandon further processing, or LQMATCH_NEXT_FILE, in which case LQT_MakeMatchesWhere will not call the AcceptFunction again until a match is found in a document with a different File Identifier (FID). ¶ A NULL AcceptFunction pointer is equivalent to one that always returns LQMATCH_ACCEPT, except much more efficient. ❦ Returns: The number of matches accepted. All matches that are accepted are stored in the given Phrase object. ❦ See Also: LQT_StringToPhrase‡ (p. 22).

---

Retrieval/Matching,
Retrieval/Phrases
../liblqtext/query.c

API t_LQT_Query *
**LQT_ParseQuery**(db, theString)
```
t_LQTEXT_Database *db;
char *theString;
```

Parses the given string, and returns a Query object. ❦ Returns: The new Query object, or LQT_-BadQuery on error. ❦ See Also: LQT_StringTo-Phrase‡ (p. 22).

---

Retrieval/Matching,
Retrieval/Phrases
../liblqtext/apmatch.c

API int
**LQT_PrintAndAcceptOneMatch**(db, Phrase, Match)
```
t_LQTEXT_Database *db;
t_Phrase *Phrase;
t_Match *Match;
```

This is intended for use as a callback function to be passed as an argument to LQT_-MakeMatchesWhere. It prints each match to stdout as it is read from disk, and also accepts it so that it is retained in the Phrase data structure. ❦ Returns: LQM_ACCEPT ❦ See Also: LQT_StringToPhrase‡ (p. 22); LQT_PrintAndRejectOneMatch‡ (below).

---

Retrieval/Matching,
Retrieval/Phrases
../liblqtext/rpmatch.c

API int
**LQT_PrintAndRejectOneMatch**(db, Phrase, Match)
```
t_LQTEXT_Database *db;
t_Phrase *Phrase;
t_Match *Match;
```

This is intended for use as a callback function to be passed as an argument to LQT_-MakeMatchesWhere. It prints each match to stdout as it is read from disk, and also rejects it so that it is not retained in the Phrase data structure. ❦ Returns: LQM_ACCEPT ❦ See Also: LQT_StringToPhrase‡ (p. 22); LQT_PrintAndRejectOneMatch‡ (above).

ARGSUSED2*/
API void
**LQT_ResetPhraseMatch**(db, thePhrase)
  t_LQTEXT_Database *db;
  t_Phrase *thePhrase;

Resets internal pointers within a phrase so that LQT_MakeMatches can be called. This is also called by LQT_MakeMatches and LQT_-MakeMatchesWhere, and is provided so that clients can write their own phrase matching routines compatibly. ❦ See Also: LQT_StringToPhrase‡ (p. 22); LQT_MakeMatches‡ (p. 17).

Retrieval/Matching,
Retrieval/Phrases
../liblqtext/phrreset.c

*The most comnonly used lq-text functions are in this category (and also in the Database/Defaults section, strictly speaking).*

*Functions in this category deal with converting a string into an internal data structure representing a phrase, and getting a list of matches for that phrase.*

*The intermediate step involving the internal data structure allows clients to determine useful information, such as how many words were recognised in the phrase, without the overhead of doing the actual match.*

---

API void
**LQT_DestroyPhrase**(db, Phrase)
  t_LQTEXT_Database *db;
  t_Phrase *Phrase;

Retrieval/Phrases, Memory
../liblqtext/freephr.c

Frees any memory associated with the given phrase, and then frees the Phrase itself. After calling LQT_-DESTROYPHRASE, it is an error to attempt to dereference the Phrase, and the operating system may detect this and raise an exception or send a fatal signal. ❦ Notes: LQT_DESTROYPHRASE does not follow the Next element of the given Phrase. A caller doing this should take a copy of Phrase→Next before calling LQT_-DESTROYPHRASE, as after the call the pointer itself will be inaccessible. ❦ See Also: LQT_STRINGTOPHRASE‡ (overleaf).

---

ARGSUSED2*/
API int
**LQT_NumberOfWordsInPhrase**(db, Phrase)
  t_LQTEXT_Database *db;
  t_Phrase *Phrase;

Retrieval/Phrases
../liblqtext/phrnword.c

Returns the number of recognised words in the phrase. Common words, or other things that the various LQT_READWORD functions would skip, are not included in the count. A phrase containing no recognised words can never be matched. ❦ Returns: the number of words in the phrase. ❦ See Also: LQT_READWORDFROMSTRINGPOINTER‡ (p. 15).

**LQT_ReadWordFromStringPointer** Database/Retrieval, Database/Documents, p. 15

**LQT_StringToPhrase** Retrieval/Phrases, p. 22

Retrieval/Phrases
../liblqtext/phſtring.c

API char *
**LQT_PhraseToString**(db, Phrase)
  t_LQTEXT_Database *db;
  t_Phrase *Phrase;

Returns a ſtring representation of a phrase.</P>. ¶ This can be used for tracing, or to give users feedback about how a phrase query was interpreted. ❦ Returns: a pointer to a freshly malloc'd ſtring, which the caller should free. ❦ See Also: LQT_StringToPhrase[‡] (below).

---

Retrieval/Phrases
../liblqtext/phſtring.c

API t_Phrase *
**LQT_StringToPhrase**(db, String)
  t_LQTEXT_Database *db;
  char *String;

Creates a data ſtructure representing the natural language phrase contained in the given String. ¶ Words in the phrase that could not possibly be in the index are not included in the ſtructure. This could be because they are in the ſtop liſt or are too short, or because the IndexNumbers parameter is set to 'off' in the database configuration file and the words begin with a digit. ¶ Words that could be in the database, but are not, are also excluded, but in this case the phrase cannot of course be matched. ¶ Words ending in * or ? are considered to be wildcards; they are expanded automatically by LQT_-MakeMatchesWhere, or you can use LQT_ExpandWildCard to iterate over all the matches. ¶ You can use LQT_NumberOfWordsInPhrase on the returned result, if it is not NULL, to determine the number of words in the ſtring that were recognised as words that are in the database. ¶ The result of LQT_StringToPhrase can be passed to LQT_-MakeMatches to find all occurrences of the phrase in the database.</P>. ❦ Returns: the created t_Phrase, or NULL if either an error occurred or there were no recognised words in the given String. ❦ See Also: LQT_MakeMatchesWhere[‡] (p. 17); LQT_DestroyPhrase[‡] (previous page).

---

Retrieval/Phrases
../liblqtext/phrunkw.c

ARGSUSED2*/
API int
**LQT_UnknownWordsInPhrase**(db, Phrase)
  t_LQTEXT_Database *db;
  t_Phrase *Phrase;

Returns the number of unrecognised words in the given phrase. A phrase containing any unrecognised words can never be matched. ❦ Notes: This number is not included in the result of LQT_NumberOfWordsInPhrase. ❦ Returns: the number of unrecognised words in the phrase.

**LQT_DestroyPhrase** Retrieval/Phrases, Memory, p. 21
**LQT_MakeMatchesWhere** Retrieval/Matching,
  Retrieval/Phrases, p. 17
**LQT_StringToPhrase** Retrieval/Phrases, p. 22

## Error Handling

*Currently, the lq-text library uses a fairly simpliſtic error handling policy that can result in calls to the syſtem call <var>exit</var>. The function* LQE_Error *is called with an argument indicating the severity of the error, combined with bitwise 'or' with any of a number of flags.*

*In addition, there are a number of wrappers for syſtem calls that are integrated with the error handling mechanism. These routines perform in exactly the same way as the corresponding syſtem calls or library functions if there are no errors, but, in the event of an error, call* LQE_Error *with a much clearer message than (for example) 'perror' would generate.*

Error Handling
../liblqerror/error.c

```
void
Error(Severity, format, a, b, c, d, e, f, g, h)
  unsigned int Severity;
  CONST char *format;
  int a, b, c, d, e, f, g, h;
```

Prints an error message, treating the given format argument as a printf-ſtyle format. The remaining arguments are optional, as for printf. ¶ The error message is prepended by the command name (using the cmdname global variable, if set, or the value of the $CMDNAME environment variable otherwise), the program name (using the value of the global 'progname', assigned by LQT_InitFromArgv from argv[0] if not already set), and a ſtring denoting the severity of the error, as determined by the Severity argument. ¶ The Severity argument is a combination using bitwise or of the values defined in <error.h>, of which the moſt commonly used are as follows: ¶ E_FATAL, which makes Error call exit and terminate the program;</P>. ¶ E_WARN, which makes Error print 'warning: ', and does not call exit; ¶ E_BUG, used on an assertion failure or on detecting a severe problem that should be caught by teſting; if any trace flags are set, E_BUG makes Error call abort to generate a core dump. ¶ E_MEMORY; you should always include this if you think it might not be safe to call malloc, for example because the heap is corrupted or there is no more free memory. ¶ E_SYS, which indicates a failed syſtem or library call, and makes Error print the corresponding syſtem error message using errno; be warned that on moſt syſtems, printf and other ſtdio functions may cause errno to be set even when there is no error, since they call isatty, which sets errno as a side-effect. ¶ E_INTERNAL, which makes Error prepend the message with the ſtring 'internal error: '; ¶ E_MULTILINE, which should be used on all lines of a multi-line error message where Error is called multiple times; the laſt call to Error in the sequence muſt include the E_-LASTLINE flag; ¶ E_LASTLINE, which is only ever used on the laſt of a sequence of several successive calls to Error to build up a single message that spans several lines; in the case of E_FATAL errors, it is only on this call that Error will call exit, for example. ❦ Bugs: An embedded newline in a string will cause a core dump on some systems. Error appends a newline automatically, so the safest thing to do is to omit the newline.

# *Tracing*

*The lq-text tracing mechanism may seem a little complex at first, because it provides a rich* API*. The functions bear close investigation, as they are used extensively within all of the lq-text code, and are designed to be very efficient.*

*You can turn on or off tracing on any of a large number of features separately, using symbolic constants or string names.*

*The* LQT_Trace *function has an interface similar to 'printf', and is thus very straight forward to use.*

*The macro* LQT_TraceFlagsSet *is very efficient and can be used in a test to surround code that is only used when debugging or when providing more verbose progress messages than usual.*

---

Tracing
../liblqtext/prflags.c

API char *
**LQT_FlagsToString**(Flags, WordFlagNamePairArray, Separator)
  unsigned long Flags;
  t_FlagNamePair *WordFlagNamePairArray;
  char *Separator;

Returns a printable string representation of the given flags, primarily intended for humans to read. The WordFlagNamePairArray argument is an array of (Name, Value) pairs; for each such pair, if all set bits in Value are also set in the Flags argument passed to LQT_FlagsToString, the corresponding Name string is appended to the result. The array is terminated by a pair with a null Name pointer; this is used rather than a count so that the array can be initialised at compile time. ¶ Adjacent Names in the result are separated with the given separator. If the flags are zero, the array is searched for a Value of zero, and, if one is found, the corresponding Name is used; to have a zero value return an empty string, use a pair with Name pointing to a zero-length string, not a null pointer. If no zero Value is found, the string "none" is used instead. ❦ Returns: a pointer to a statically allocated string. ❦ Errors: Passing null or invalid values for WordFlagNamePairArray or Separator will cause unpredictable results. There must be enough memory to allocate the result, which grows automatically as needed, but never shrinks. ❦ See Also: LQT_StringToFlags‡ (p. 28).

---

Tracing
../liblqtext/trace.c

API int
**LQT_ForEachTraceFlag**(CallMe)
```
  void (* CallMe)(
    char *Name,
    unsigned int Value,
    int isSet
  );
```

Calls the given function for each available trace flag. The integer argument IsSet passed to the function is non-zero for those flags that are set in the current trace flags, and zero for the others. ¶ The flags are defined in the `<lqtrace.h>` header file. ❦ Returns: zero. ❦ See Also: LQT_Trace‡ (p. 28); LQT_SetTraceFlag‡ (opposite).

---

Tracing
../liblqtext/trace.c

API char *
**LQT_GetGivenTraceFlagsAsString**(Flags)
```
  t_TraceFlag Flags;
```

This function works like LQT_GetTraceFlagsAsString, except that it uses the given flags instead of the current value of the lq-text trace flags. ¶ The caller should not attempt to write into, or free, the result string. ¶ The flags are defined in the `<lqtrace.h>` header file. ❦ Returns: non-zero if one or more flags satisfies the constraints ❦ Notes: You can get a rather long line giving all possible flags using the C expression (t_TraceFlag) ˜(unsigned long) 0, which provides a number with all bits set, as an argument to LQT_GetGivenTraceFlagsAsString. ❦ See Also: LQT_Trace‡ (p. 28); LQT_SetTraceFlag‡ (opposite); LQT_GetTraceFlags‡ (below); LQT_GetTraceFlagsAsString‡ (opposite).

---

Tracing
../liblqtext/trace.c

API t_TraceFlag
**LQT_GetTraceFlags**()

Returns the current value of the lq-text trace flags. The various flag values are defined in the `<lqtrace.h>` header file, and may be combined (using bitwise or) in any combination. ¶ The value returned by LQT_GetTraceFlags should not normally be used by itself in diagnostic or error messages. Instead, use LQT_GetTraceFlagsAsString, which provides a more readable value for humans. ❦ Returns: the current lq-text trace flags, or'd together ❦ See Also: LQT_Trace‡ (p. 28); LQT_SetTraceFlag‡ (opposite); LQT_UnSetTraceFlag‡ (p. 29); LQT_SetTraceFlagsFromString‡ (opposite).

API char *
**LQT_GetTraceFlagsAsString**()

Tracing
../liblqtext/trace.c

Returns a ſtatic pointer to a ſtring representation of the current lq-text trace flags. This is suitable for printing in error messages, and can also be used with LQT_SetTraceFlagsFromString to save and reſtore flags in a machine-independent way. ¶ The caller should not attempt to write into, or free, the result ſtring. ¶ The flags are defined in the `<lqtrace.h>` header file. ❦ Returns: a pointer to a private ſtring. ❦ See Also: LQT_Trace‡ (overleaf); LQT_SetTraceFlag‡ (below); LQT_UnSetTraceFlag‡ (p. 29); LQT_GetGivenTraceFlagsAsString‡ (opposite); LQT_SetTraceFlagsFromString‡ (below).

---

API FILE *
**LQT_SetTraceFile**(newFile)
  FILE *newFile;

Tracing
../liblqtext/trace.c

After this call, all lq-text tracing output produced with LQT_Trace will be sent to the given file. It is the caller's responsibility to ensure that the given FILE * is valid and points to a file that is open for writing. ¶ The default file used before LQT_SetTraceFile has been called is ſtderr. An argument of (FILE *) NULL will reset the file to the default value, but will not close the given ſtream. The file is also not when a database is closed; see LQT_AddActionOnClose for a way of changing this behaviour. ❦ Returns: the previous file pointer ❦ See Also: LQT_Trace‡ (overleaf); LQT_SetTraceFlag‡ (below); LQT_AddActionOnClose‡ (p. 3).

---

API t_TraceFlag
**LQT_SetTraceFlag**(theFlag)
  t_TraceFlag theFlag;

Tracing
../liblqtext/trace.c

Adds the given argument to the current lq-text trace flags. You can add several flags at a time by combining them with bitwise or. If you do, the return value may be hard to decipher, although since the return value is primarily of intereſt to internal liblqtext routines, this probably doesn't matter. ❦ Returns: non-zero if any of the the given flags were set. ❦ See Also: LQT_Trace‡ (overleaf); LQT_UnSetTraceFlag‡ (p. 29); LQT_GetTraceFlags‡ (opposite).

---

API char *
**LQT_SetTraceFlagsFromString**(theString)
  char *theString;

Tracing
../liblqtext/trace.c

Attempts to set the lq-text trace flags by reading a ſtring representation of them. The ſtring muſt be in the format produced by LQT_GetTraceFlagsAsString; in other words, a sequence of words separated by the vertical bar. The various flag values are defined in the `<lqtrace.h>` header file, and may be combined (using bitwise or) in any combination ¶ If the return value points to a NUL byte, the end of the ſtring was reached without error; otherwise, it is up to the caller to determine whether the extra unconverted text was expected. ❦ Returns: a pointer to the firſt unconverted character in the given ſtring ❦ See Also:

**LQT_AddActionOnClose** Database/Database, p. 3
**LQT_GetGivenTraceFlagsAsString** Tracing, p. 26
**LQT_GetTraceFlags** Tracing, p. 26
**LQT_SetTraceFlag** Tracing, p. 27
**LQT_SetTraceFlagsFromString** Tracing, p. 27
**LQT_Trace** Tracing, p. 28
**LQT_UnSetTraceFlag** Tracing, p. 29

LQT_Trace‡ (overleaf); LQT_SetTraceFlag‡ (previous page); LQT_GetTraceFlagsAsString‡ (previous page); LQT_StringToFlags‡ (below).

---

Tracing
../liblqtext/rdflags.c

API char *
**LQT_StringToFlags**(String, Flagp, WordFlagNamePairArray, Separator)
  char *String;
  unsigned long *Flagp;
  t_FlagNamePair *WordFlagNamePairArray;
  char *Separator;

Tries to reverse the operation of LQT_FlagsToString. In other words, LQT_StringToFlags takes a ſtring which it assumes to be a sequence of names of flags found in the given FlagNames array, separated by the given conſtant ſtring, and returns the bitwise 'or' of the Value members corresponding to the Names that are found. ¶ In addition, a leading + or - is used to indicate that the following flags are to be added (with bitwise or) or removed (usinbg bitwise and on their negation) from the result. ❦ Returns: a pointer to the firſt unconverted character in String, and the actual value in Flagp ❦ See Also: LQT_StringToWordFlags‡ (below); LQT_WordFlagsToString‡ (opposite).

---

Tracing
../liblqtext/rdflags.c

API char *
**LQT_StringToWordFlags**(db, String, Flagp)
  t_LQTEXT_Database *db;
  char *String;
  unsigned long *Flagp;

Tries to reverse the operation of LQT_WordFlagsToString. In other words, LQT_StringToWordFlags takes a ſtring which it assumes to be a sequence of names of flags as defined in the header file `<wordrules.h>` separated by LQTpWordFlagSep (a comma), and returns the bitwise 'or' of the Word Flags corresponding to the Names that are found. ❦ Returns: a pointer to the firſt unconverted character in String, and the actual value in Flagp ❦ See Also: LQT_StringToWordFlags‡ (above); LQT_WordFlagsToString‡ (opposite).

---

Tracing
../liblqtext/trace.c

API void
**LQT_Trace**(Flags, Format, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o)
  t_TraceFlag Flags;
  char *Format;

Prints diagnoſtic messages. The Flags argument muſt be one or more flags taken from `<lqtrace.h>` and combined with bitwise or. If one of more of the given Flags is set in the current lq-text trace flags, the remainder of the arguments are passed to fprintf ¶ For efficiency, it may be beſt to use LQT_TraceFlagsSet firſt to determine whether to call LQT_Trace, as the former is likely to be implemented as a short macro in `<lqtrace.h>`, but currently LQT_Trace cannot be so implemented. ¶ Each line of trace output is preceded by the current program name, the word 'trace', and a ſtring representation of one or more of those flags in the Flags argument to LQT_Trace which are set in the current lqtext trace flags. ❦ See Also: LQT_SetTraceFlag‡ (p. 27); LQT_UnSetTraceFlag‡ (below); LQT_SetTraceFlagsFromString‡ (p. 27).

---

LIBRARY int
**LQTp_TraceFlagsSet**(QueryFlags)
  t_TraceFlag QueryFlags;

Tracing
../liblqtext/trace.c

Determines whether any of a particular group of trace flags are set; if so, a non-zero value is returned, otherwise zero. The flags may have been or'd together, and are defined in the `<lqtrace.h>` header file.

For each such flag, all of the bits set in the flag have been be set in the argument to LQT_-TraceFlags in order for it to be considered as being set. ❦ Returns: non-zero if one or more flags satisfies the constraints ❦ Notes: This may be implemented as a macro; the prototype shown may in that case have a different name. ❦ See Also: LQT_Trace‡ (opposite); LQT_SetTraceFlag‡ (p. 27); LQT_UnSetTraceFlag‡ (below); LQT_GetTraceFlags‡ (p. 26).

---

API int
**LQT_UnSetTraceFlag**(theFlag)
  t_TraceFlag theFlag;

Tracing
../liblqtext/trace.c

The given flag is removed from the current lq-text trace flags. You can combine multiple flag values using bitwise or. ¶ This routine can be used in conjunction with LQT_GetTraceFlags to unset all of the current flags. ❦ Returns: 1 if any of the given flags were set, and 0 otherwise. ❦ Errors: An attempt to unset flags that were not set produces an error of type E_WARN|E_-INTERNAL. ❦ See Also: LQT_Trace‡ (opposite); LQT_GetTraceFlags‡ (p. 26); LQT_SetTraceFlagsFromString‡ (p. 27).

---

API char *
**LQT_WordFlagsToString**(db, Flags)
  t_LQTEXT_Database *db;
  t_WordFlags Flags;

Tracing
../liblqtext/prflags.c

Returns a printable string representation of the given WordFlags, intended for humans to read as well as for use with the LQT_WordFlagsToString function. The flags are defined in `<wordrules.h>` and are explained under the Language category.

❦ Returns: a pointer to a statically allocated string. ❦ Errors: There must be enough memory to allocate the result, which grows automatically as needed, but never shrinks. ❦ See Also: LQT_StringToWordFlags‡ (opposite); LQT_FlagsToString‡ (p. 25).

**LQT_FlagsToString** Tracing, p. 25
**LQT_GetTraceFlags** Tracing, p. 26
**LQT_SetTraceFlag** Tracing, p. 27
**LQT_SetTraceFlagsFromString** Tracing, p. 27
**LQT_StringToWordFlags** Tracing, p. 28
**LQT_Trace** Tracing, p. 28
**LQT_UnSetTraceFlag** Tracing, p. 29

## *Input*

*Currently, this category is thinly populated; it contains a useful routine for reading a match in the format produced by all the lq-text clients, and returning a data ſtructure that can be used with other functions.*

*The Output category contains a routine that does the inverse operation, taking the data ſtructure and printing the information to a ſtdio ſtream.*

---

Input
../liblqtext/rmatch.c

API char *
**LQT_StringToMatch**(db, Severity, theString, theMatchpp)

```
  t_LQTEXT_Database *db;
  int Severity;
  char *theString;
  t_MatchStart **theMatchpp;
```

Converts a ſtring representation of a match to a t_-Match object. Leading and trailing white space on the line is ignored. ¶ The match is considered to consiſt of a number of ASCII decimal numbers followed by a file name. The numbers are, in this order, the number of words matched, the block within the file, the word within the block, and the File Identifier (t_FID). There may be an optional filename after the FID. ¶ If the FID is given as zero, there muſt be a filename, and this is given as an argument to LQT_NameToFID to complete the FID entry in the match. ❦ Notes: The returned Match is contained in a ſtatic buffer and should not be freed or overwritten. You muſt make a copy if you need to retain the information over successive calls to LQT_StringToMatch. The FileName field of the Match will point either into the middle of the given ſtring, or to an internal ſtatic buffer, or, in the case that the given FID was invalid, will be NULL ¶ A ſtatic internal buffer is retained containing the previous result of LQT_NameToFID, for efficiency in the common case that there are several matches in a row from the same document. ❦ Returns: 1. NULL if there was no error; in that case, *theMatchpp is set to either a pointer to a Match, or NULL if the line didn't contain a match. 2. On error, a ſtring describing the problem is returned.

## *Output*

*This category contains a routine for printing a match in the same format as the lq-text clients. It is typically several times faſter than using printf.*

---

API void
**LQT_fPrintOneMatch**(db, theFile, FirſtNumber, FileInfo, WordPlace)

```
  t_LQTEXT_Database *db;
  FILE *theFile;
  int FirſtNumber;
  t_FileInfo *FileInfo;
  t_WordPlace *WordPlace;
```

Prints a single match to the given ſtdio file descriptor, in a form that will subsequently be underſtood by the routines and programs that read matches. ❦ Notes: Not all the information ſtored in the database index for each match is printed by LQT_fPrintOneMatch. A future release will allow you to change the print format (using a Name Space).

*¶ The term 'Documents' is used to refer to the files that have been indexed, as opposed to the files that make up the actual database. Functions in the Database/Documents category thus deal with accessing the indexed documents.*

---

API void
**LQT_DestroyFileInfo**(db, FileInfo)
  t_LQTEXT_Database *db;
  t_FileInfo *FileInfo;

Database/Documents, Memory
../liblqtext/fileinfo.c

Frees the memory used by the given FileInfo. Neither the database nor the file described by the FileInfo is affected; LQT_DestroyFileInfo frees any internal data structures associated with the FileInfo and then frees the FileInfo itself. After calling LQT_DestroyFileInfo, the FileInfo pointer is no longer valid, and should not be dereferenced. ❦ See Also: LQT_NameToFID‡ (p. 14); LQT_FIDToFileInfo‡ (p. 9).

---

API char *
**LQT_FindFile**(db, Name)
  t_LQTEXT_Database *db;
  char *Name;

Database/Documents
../liblqtext/docpath.c

Returns a pointer to a full pathname, given a filename as stored in the lq-text File index. The current database DocPath is searched, and if that fails, an attempt is made to find the file with .gz appended, then with .Z appended. ¶ The returned string points to a static buffer, and should not be freed. The buffer is overwritten on successive calls to LQT_FindFile(). ❦ Bugs: Does not understand the archive name notation, archive(filename).

---

LIBRARY char *
**LQT_GetFilterName**(db, FileInfo)
  t_LQTEXT_Database *db;
  t_FileInfo *FileInfo;

Database/Documents
../liblqtext/filters.c

Returns a short name describing the file type associated with the given file. The value is static, and should not be freed by the caller. ❦ See Also: LQT_GetFilterType‡ (overleaf).

**LQT_FIDToFileInfo** Database/Retrieval, Database/Documents, p. 9
**LQT_GetFilterType** Database/Documents, p. 36
**LQT_NameToFID** Database/Retrieval, Database/Documents, p. 14

Database/Documents
../liblqtext/filters.c

LIBRARY int
**LQT_GetFilterType**(db, FileInfo, StatBuf)
  t_LQTEXT_Database *db;
  t_FileInfo *FileInfo;
  ſtruct ſtat *StatBuf;

Determines the appropriate filter to use to read the file represented by the given FileInfo; this is an internal routine and will be replaced in the next release. ❦ See Also: LQT_UnpackAndOpen‡ (below).

---

Database/Documents
../liblqtext/getfid.c

API t_FID
**LQT_GetMaxFID**(db)
  t_LQTEXT_Database *db;

Returns the largeſt allocated FID. ❦ Returns: 1. the largeſt FID already allocated. 2. 1 if no FIDs have been allocated. ❦ Errors: as for LQT_ReadBlock, LQT_OpenDataBase.

---

Database/Documents
../liblqtext/setfid.c

API t_FID
**LQT_GetMaxOrAllocateFID**(db, WriteCurrent)
  t_LQTEXT_Database *db;
  int WriteCurrent;

Allocates a new FID, and writes the new value to disk. If the 'WriteCurrent' argument is zero, the value is only written in one on every 1,000 calls. ❦ See Also: LQT_SyncDatabase‡ (p. 5).

---

Database/Documents
../liblqtext/unpack.c

API int
**LQT_UnpackAndOpen**(db, FileName)
  t_LQTEXT_Database *db;
  char *FileName;

Tries to open the named file, using compress or gunzip as necessary. Can append a .Z or .gz to the file name. Currently, LQT_UnpackAndOpen makes a copy of a file if necessary; a future version may create a pipe, and the interface will change. ❦ Returns: 1. an open file descriptor on success; 2. −1 if the file couldn't be opened. ❦ See Also: LQT_FindFile‡ (previous page); LQT_MakeInput‡ (p. 51).

---

Database/Documents
../liblqtext/setfid.c

API void
**LQT_WriteCurrentMaxFID**(db)
  t_LQTEXT_Database *db;

Writes the cached value of the largeſt allocated FID to disk. This routine is regiſtered by LQT_-OpenDatabase so that it is called automatically by LQT_CloseDatabase. ¶ It may also be useful to call it directly for the purpose of debugging a new lq-text client that updates the database, for example when running under a database. ❦ See Also: LQT_WriteCurrentMaxWID‡ (p. 56); LQT_SyncDatabase‡ (p. 5); LQT_CloseDatabase‡ (p. 4).

**LQT_CloseDatabase** Database/Database, p. 4
**LQT_FindFile** Database/Documents, p. 35
**LQT_MakeInput** Database/Update, Database/Documents, p. 51
**LQT_SyncDatabase** Database/Database, p. 5
**LQT_UnpackAndOpen** Database/Documents, p. 36
**LQT_WriteCurrentMaxWID** Database/Words, p. 56

## *Database/Dynamic Hashing*

*Functions in this category are related to manipulating the dynamic hashing database that lq-text relies upon. A dynamic hashing database provides a key to value mapping; the key can be any binary data, and so can the value.*

*Two dynamic hashing databases are used by lq-text: the first is used to map a word into a* WID*, that is, into a Word IDentifier number. The second is used to map a filename into a* FID*, that is, into a File IDentifier.*

*You can configure lq-text to use any of a number of different dynamic hashing packages; ndbm is supplied with most Unix systems; Berkeley's 'db' package is included with lq-text, along with Ozan Yigit's 'sdbm' package. Whichever package you use, the result is essentially the same, except that some packages are faster or more reliable than others. For large databases (say, several hundred megabytes), you will probably need to use the db package, since it has fewer size limits than most others.*

*The individual dynamic hashing packages provide documentation on the various routines, such as* DBM_FETCH *and* DBM_STORE*, that you can use with the databases. The lqword sample client uses routines that iterate over all entries in a database, one by one.*

---

ARGSUSED*/
API int
**LQT_CloseKeyValueDatabase**(db)
  DBM *db;

Database/Dynamic Hashing, Database/Files
../liblqtext/smalldb.c

This currently does nothing, since the Key Value Databases are kept open. If the library is compiled with dbm instead of ndbm, or with the cache disabled, LQT_CloseKeyDatabase becomes active, so it should be paired with every call to LQT_-OpenKeyValueDatabase ❦ See Also: LQT_SyncAndCloseAllKeyValueDatabases‡ (overleaf).

**LQT_SyncAndCloseAllKeyValueDatabases** Database/Dynamic Hashing, Database/Files, p. 38

Database/Dynamic Hashing, Database/Files ../liblqtext/smalldb.c

API DBM *
**LQT_OpenKeyValueDatabase**(db, FilePrefix)

```
  t_LQTEXT_Database *db;
  char *FilePrefix;
```

Opens an ndbm-style database of the given name, creating it if the current database modes allow it. The function keeps a cache of open databases, so that if there is already an open database of the given name, its handle is simply returned. ¶ Opening a Key Value Database involves several file system accesses and using malloc to obtain memory, so it's much better to use the cached values. It is even better still to keep frequently used Key Value Databases open, for example in a static variable, and to close them only when the database is closed. ❦ Returns: A handle (usually a DBM * pointer) to the named Key Value Database. ❦ Errors: If the underlying ndbm-style database couldn't be opened, a fatal error is produced (E_FATAL|E_SYS) indicating the problem. One possible cause of this is that $HOME/LQTEXTDIR isn't a directory, or doesn't exist, and $LQTEXTDIR isn't set to point to a suitable alternate directory. Another possible problem is that a previous run of lqaddfile failed, and left the Key Value Databases locked for writing; the best thing to do in this case is to run the lqclean program and start again. ❦ See Also: LQT_CloseKeyValueDatabase‡ (previous page); LQT_OpenDatabase‡ (p. 5); LQT_AddActionOnClose‡ (p. 3); LQT_SyncDatabase‡ (p. 5).

---

Database/Dynamic Hashing, Database/Files ../liblqtext/smalldb.c

API int
**LQT_SyncAndCloseAllKeyValueDatabases**(db)

```
  t_LQTEXT_Database *db;
```

Closes all Key Value Databases that have been opened, after writing any pending data to disk. ¶ This function is registered automatically as an action to be performed when a database is closed or on a call to LQT_Sync, and should not normally need to be called directly. The return value and argument are for compatibility with LQT_AddActionOnClose. The argument must be a null pointer, for future compatibility. ❦ See Also: LQT_OpenKeyValueDatabase‡ (above); LQT_AddActionOnClose‡ (p. 3); LQT_CloseDatabase‡ (p. 4).

---

Database/Dynamic Hashing, Database/Files ../liblqtext/smalldb.c

LIBRARY char *
**LQTp_CreateEmptyKeyValueDatabase**(db, Directory, prefix)

```
  t_LQTEXT_Database *db;
  char *Directory;
  char *prefix;
```

Some versions of dbm or ndbm provided with various Unix systems do not automatically create a new DBM file, even when asked to; it is necessary to create the file with the open(2) or creat(2) system calls. The original Unix dbm library was like this. ¶ This function creates the necessary files, in the given Directory; the files will have names beginning with the given Prefix, and

**LQT_AddActionOnClose** Database/Database, p. 3
**LQT_CloseDatabase** Database/Database, p. 4
**LQT_CloseKeyValueDatabase** Database/Dynamic Hashing, Database/Files, p. 37
**LQT_OpenDatabase** Database/Database, p. 5
**LQT_OpenKeyValueDatabase** Database/Dynamic Hashing, Database/Files, p. 38
**LQT_SyncDatabase** Database/Database, p. 5

depending on the version of ndbm in use, may have a suffix such as .db; BSD db uses a single file, but moﬅ other implementations use two, one called Prefix.dir and one called Prefix.pag. ¶ This routine is called automatically by LQT_OpenKeyValueDatabase when necessary, but is made available for general use for convenience. ❦ Bugs: LQTp_-CreateEmptyKeyValueDatabase should be in liblqutil inﬅead. ❦ See Also: LQT_OpenKey-ValueDatabase‡ (opposite).

*The term 'Documents' is used to refer to the files that have been indexed, as opposed to the files that make up the actual database. Functions in the Database/Files category thus deal with accessing and manipulating the files in the lq-text database directory (\$LQTEXTDIR).*

*The functions in this category are at a low level; usually, there is a higher level routine that will do what you want, unless you are modifying the internals of liblqtext, or are writing complex database update code.*

*Note that the category Utilities/Files also exists, and provides functions such as* LQU_IsDir *for determining whether a given string is the name of a directory.*

---

API int
**LQT_BlockIsCached**(db, Block)
  t_LQTEXT_Database *db;
  unsigned long Block;

Database/Files, Database/Physical ../liblqtext/pbcache.c

Determine whether the block at a given offset in the data file is in the block buffer cache or not. Since LQT_ReadBlock returns a pointer into the cache, it is a fatal error (E_BUG) if LQT_WriteBlock is called for a block that is not cached. ¶ The cache is always large enough to hold at least the last two blocks returned by LQT_ReadBlock. This is just enough to ensure that the NextOffset field in a block's header can be filled in after allocating the next block in a chain. ❦ Returns: Non-zero if the block is cached, and zero otherwise. ❦ Errors: Fatal error if the main data file can't be opened or created. ❦ Notes: As a side-effect, the CurrentBlock variable in pbcache.c is set to point to the cached block; this is used internally by the library routines in that file. ❦ See Also: LQT_ReadBlock‡ (overleaf); LQT_WriteBlock‡ (p. 43).

Database/Files, Database/Physical ../liblqtext/pbcache.c

LIBRARY int
**LQT_FlushBlockCache**(db)
  t_LQTEXT_Database *db;

Writes any pending dirty blocks to the disk. Copies of the blocks are retained in memory, however, until LQT_CloseDatabase is called, and will be found by LQT_BlockIsCached and hence by LQT_ReadBlock if an attempt is made to read them again. ¶ When a database is opened, LQT_OpenDatabase adds LQT_FlushBlockCache as an action to be performed automatically whenever the database is flushed or closed. It should not be necessary to call this code directly from outside the library, and it is made available primarily to aid in debugging. ❦ Errors: Fatal error (E_BUG) if the cache is dirty in read-only mode. ❦ See Also: LQT_AddActionOnClose‡ (p. 3); LQT_OpenDatabase‡ (p. 5); LQT_SyncDatabase‡ (p. 5); LQT_CloseDatabase‡ (p. 4).

---

Database/Files, Database/Physical ../liblqtext/pbcache.c

LIBRARY int
**LQT_FlushOneBlockFromCache**(db)
  t_LQTEXT_Database *db;

If there any data blocks that are waiting to be written out to disk, LQT_FlushOneBlockFromCache will write one of them out. ¶ This function is internal to lq-text and users of the Physical layer of the database. ❦ Errors: Fatal error (E_BUG) if the cache is dirty in read-only mode. ❦ See Also: LQT_FlushBlockCache‡ (above).

---

Database/Files, Database/Physical ../liblqtext/pbcache.c

API unsigned char *
**LQT_ReadBlock**(db, Offset, WID)
  t_LQTEXT_Database *db;
  unsigned long Offset;
  t_WID WID;

Reads the block at the given byte offset, and returns a pointer to the data. The data is stored in a cache, so it is important not to try and write beyond the end of the block or group of blocks as determined by LQT_ExtendBlock or LQT_FindFreeBlock. The block must be written out with LQT_WriteBlock if it has changed. In addition, the block is not locked in memory, but LQT_ReadBlock ensures that it is safe to read at least one other block before writing this one out with LQT_WriteBlock. ❦ Returns: A pointer to the data ❦ Notes: Attempts to read beyond the end of the data file will extend the database automatically. The data will be initialised to zero, except for the block headers, whose NumberOfBlocks field will all be set to one. ❦ Errors: Fatal error (E_BUG) if the database can't be opened or created. ❦ See Also: LQT_ExtendBlock‡ (p. 46); LQT_FindFreeBlock‡ (p. 46); LQT_WriteBlock‡ (opposite).

---

Database/Files, Database/Defaults ../liblqtext/filtertb.c

API int
**LQT_ReadFilterTable**(db)
  t_LQTEXT_Database *db;

Reads the filter table into memory if one was specified. The filter table lists the file types that can be indexed, and gives an index filter for each of them. ¶ If there is no 'filtertable' entry in the database configuration file, a built-in list of defaults is used. ❦ Returns: zero on success. ❦ See Also:

**LQT_AddActionOnClose** Database/Database, p. 3
**LQT_CloseDatabase** Database/Database, p. 4
**LQT_ExtendBlock** Database/Physical, p. 46
**LQT_FindFreeBlock** Database/Physical, p. 46
**LQT_FlushBlockCache** Database/Files, Database/Physical, p. 42
**LQT_OpenDatabase** Database/Database, p. 5
**LQT_SyncDatabase** Database/Database, p. 5
**LQT_WriteBlock** Database/Files, Database/Physical, p. 43

LQT_GetFilterType‡ (p. 36).

---

Database/Files, Database/Physical ../liblqtext/pbcache.c

API void
#ifdef ASCIITRACE
**LQTp_WriteBlock**(db, theFile, theLine, Block, Data, Length, theWID)
  t_LQTEXT_Database *db;
  char *theFile;
  int theLine;
#else
**LQT_WriteBlock**(db, Block, Data, Length, theWID)
  t_LQTEXT_Database *db;
#endif
  unsigned long Block;
  unsigned char *Data;
  int Length;
  t_WID theWID;

Writes the given block to the database. Actually the block is saved in the cache, and if it was originally obtained with LQT_ReadBlock it's already in the cache, so LQT_-WriteBlock simply marks it as dirty, needing to be saved. If you change data in a block without calling LQT_WriteBlock, the changes usually won't be written to disk (unless an adjacent block in the cache is written). ¶ The block must have a valid header; if the block's length field is larger than the Length argument, the extra blocks are marked as free. The header is described in `<blkheader.h>`. ❦ Errors: Format or consistency errors are generally fatal. Attempting to write a block not in the cache will produce a warning. ❦ See Also: LQT_FindFreeBlock‡ (p. 46); LQT_ReadBlock‡ (opposite); LQT_WriteBlock‡ (above).

---

Database/Files ../liblqtext/lqwverno.c

API void
**LQT_WriteVersionToDatabase**(db)
  t_LQTEXT_Database *db;

Writes the liblqtext library version to the database so that LQT_CheckDatabaseVersion will accept it. This routine is called automatically when a new lq-text database is created. ❦ See Also: LQT_CheckDatabaseVersion‡ (p. 4).

---

Database/Files ../liblqtext/wordinfo.c

LIBRARY int
**LQTpFlushWIDCache**(db)
  t_LQTEXT_Database *db;

Writes any pending entries in the wid file cache out to disk. This must be done before closing the database or exiting the running program if any changes have been made. ¶ When a database is opened, LQTpFlushWIDCache is registered as an action to be performed on an LQT_-CloseDatabase or LQT_SyncDatabase, so it should not be necessary to call this function directly. ¶ The ignored argument is for compatibility with LQT_AddActionOnClose, as is the return value. ❦ See Also: LQT_SyncDatabase‡ (p. 5); LQT_CloseDatabase‡ (p. 4);

LQT_AddActionOnClose‡ (p. 3).

*Routines in this category manipulate the database at the raw file level; they deal with data blocks on the hard disk, or with ſtreams of raw bytes.*

*The compressed numbers package is also included in this category; although it is in principle useful outside of lq-text, and has in fact been used elsewhere several times in the paſt, the routines in h/numbers.h and liblqtext/numbers.c usually need to be modified, as they are very low level.*

*You should be warned that modifying the source of any of these routines, or using them in any way incorrectly, is likely to lead to corruption in the databases you create or manipulate: the integrity of an lq-text database depends heavily on these routines.*

*This documentation is intended to be enough so that you can work with exiſting code that uses these functions; you should be prepared to use the source to undersand more if you need to use them.*

---

API int
**LQT_BlockIsFree**(db, Offset)
t_LQTEXT_Database *db;
unsigned long Offset;

Database/Physical
../liblqtext/pbcache.c

Determine the ſtatus of the block at a given byte offset from the ſtart of the data overflow file (data). An external file, freeliſt, is kept in the database directory; this file uses a single bit to represent the ſtatus of each block, either in use or free. If the freeliſt file is removed, subsequent attempts to write to the database will fail. Read-only access will ſtill work unless LQTRACE_READAFTERWRITE is set, whereupon LQT_ReadBlock checks the ſtatus of each block before returning it; it is an error to attempt to read an unallocated block, although this not normally checked, for performance reasons. ❦ Returns: Non-zero if the block is available, zero if it is free ❦ Notes: The firſt few blocks are reserved for ſtoring information about the database; they are marked as used automatically whenever a database is created. ¶ The freeliſt file can be rebuilt by the lqmkfreeliſt program. ¶ The teſt program 'free' contains examples of using the Block Status functions LQT_BlockIsFree and LQT_SetBlockStatus. It can also be used to edit the contents of the freeliſt file. ❦ Errors: Fatal error if the freeliſt file could not be opened ❦ See Also:

LQT_SetBlockStatus‡ (p. 47); LQT_FindFreeBlock‡ (below); LQT_ReadBlock‡ (p. 42).

---

Database/Physical
../liblqtext/pbcache.c

API void
**LQT_ExtendBlock**(db, Offset, BlockCountp, BytesWanted)

```
  t_LQTEXT_Database *db;
  unsigned long Offset;
  unsigned int *BlockCountp;
  unsigned long BytesWanted;
```

LQT_ExtendBlock marks as many blocks as possible following the given byte Offset as being used, and increments the unsigned int pointed to by BlockCountp by the number of blocks added. The number of blocks added is such that a single contiguous stretch of data starting at the given Offset, and continuing for the number of blocks in *BlockCountp, does not cross an LQT_ReadBlock cache boundary. ¶ If the BytesWanted argument is non-zero, the total number of blocks in BlockCountp when LQT_ExtendBlock returns will not be more than one block greater than BytesWanted bytes. ❦ Notes: BlockCountp must be greater than zero on entry to LQT_ExtendBlock. ❦ Returns: The total number of blocks in *BlockCountp ❦ See Also: LQT_FindFreeBlock‡ (below); LQT_SetBlockStatus‡ (opposite).

---

Database/Physical
../liblqtext/pbcache.c

API unsigned long
**LQT_FindFreeBlock**(db, WID, BlockLengthp, BytesWanted)

```
  t_LQTEXT_Database *db;
  t_WID WID;
  unsigned int *BlockLengthp;
  unsigned long BytesWanted;
```

Allocates a block from the free list, and marks it as in use. The block is at least BLOCKSIZE bytes long, and may be longer, as contiguous free blocks are combined to make a single longer block as long as will fit in a single cache entry. If the BytesWanted argument is non-zero, the block will not be more than BLOCKSIZE bytes longer than than BytesWanted bytes. Since LQT_FindFreeBlock does not actually read the data from the disk (or cache), it is up to the caller to ensure that LQT_ReadBlock is called, and that the resulting block's header is filled in with NumberOfBlocks equal to the value that LQT_FindFreeBlock stored in BlockLengthp. ❦ Returns: the byte offset in the data file of the block allocated, and also the number of blocks allocated (in BlockLengthp). ❦ See Also: LQT_BlockIsCached‡ (p. 41); LQT_BlockIsFree‡ (previous page); LQT_SetBlockStatus‡ (opposite).

---

Database/Physical
../liblqtext/wblock.c

API void
**LQT_FlushBlock**(db, Block, ByteCount, NextOffset, LastStart, WID)

**LQT_BlockIsCached** Database/Files, Database/Physical, p. 41
**LQT_BlockIsFree** Database/Physical, p. 45
**LQT_FindFreeBlock** Database/Physical, p. 46
**LQT_ReadBlock** Database/Files, Database/Physical, p. 42
**LQT_SetBlockStatus** Database/Physical, p. 47

```
  t_LQTEXT_Database *db;
  unsigned char *Block;
  int ByteCount;
  unsigned long *NextOffset, *LastStart;
  t_WID WID;
```

Writes out the given block to the cache. This is really the same as LQT_WriteBlock, except that it is used for the last block in each chain of matches.

---

API void
**LQT_SetBlockStatus**(db, Offset, Status)

```
  t_LQTEXT_Database *db;
  unsigned long Offset;
  int Status;
```

Database/Physical
../liblqtext/pbcache.c

Set the status of the block at a given byte offset in the data file. ¶ Status must be either SET_BLOCK_AS_USED or SET_BLOCK_AS_FREE. In the former (USED) case, the block is marked as being in use, and can be brought into the cache with LQT_ReadBlock. In the latter case (FREE), the block is marked as being available for reuse. Since LQT_SetBlockStatus does not access the actual data, it does not have access to the block's length. It is therefore the caller's responsibility to call LQT_SetBlockstatus for each contiguous block when a block header's NumberOfBlock field is greater than one. ❦ Notes: This routine was called 2,785,338 times when indexing Shakespeare's complete works. To try and speed things up, LQT_SetBlockstatus performs as few checks as possible. ❦ See Also: LQT_BlockIsFree‡ (p. 45).

---

INLINE int
**LQT_sReadNumber**(Sp, Resultp, StartOfBuffer, LengthOfBuffer)

```
  unsigned char **Sp;
  unsigned long *Resultp;
  unsigned char *StartOfBuffer;
  unsigned int LengthOfBuffer;
```

Database/Physical
../h/numbers.h

Reads a number from its compressed binary representation stored the given string. The pointer pointed to by Sp is advanced to point to the first unread byte of the buffer. The retrieved number is stored in the variable pointed to by the given Resultp argument. ❦ Returns: 1. −1 if the entire number was not read, because it wasn't all included in the given string; in this case, the pointer referred to by Sp will have been advanced by the number of bytes read, but the return value is useless. 2. Zero is returned if the number was read successfully. ❦ See Also: LQT_sWriteNumber‡ (below).

---

INLINE int
**LQT_sWriteNumber**(Sp, Number, Base, Maxlen)

```
  unsigned char **Sp;
  unsigned long Number;
  unsigned char *Base;
  unsigned int Maxlen;
```

Database/Physical
../h/numbers.h

Writes a compressed binary representation of the given Number into the given string. The pointer pointed to by Sp is advanced to point to the first unwritten byte of the buffer. ❦ Returns: 1. −1 if the string doesn't fit; in this case, the pointer referred to by Sp will have been advanced by the amount of the number that fitted; 2. Zero is returned if the number was written successfully. ❦ Notes: This function and the companion LQT_sReadNumber are central to the operation of the lq-text database package. If it were not for the use of compressed numbers, the index would be too large to be useful. ¶ The function is designed to work best with small numbers; a number less than 127 is written out in a single byte, for example, and a number less than 16383 is written in two bytes. For this reason, LQT_-sWriteNumber is most effectively used when writing a sorted sequence of numbers, as then you can write only the difference between successive values, saving space. This form of delta coding is used extensively by lq-text. ❦ See Also: LQT_sReadNumber‡ (previous page).

## *Database/Update*

*These routines are for modifying an lq-text database. You may need to link against src/lqtext/wordtable.o to use some of them in the current release. See the lqaddfile client for examples of using some of them.*

---

API unsigned long
**LQT_AddWordPlaces**(db, WordPlaces, WID, Offset, NumberToWrite)

```
  t_LQTEXT_Database *db;
  t_WordPlace *WordPlaces;
  t_WID WID;
  unsigned long Offset;
  unsigned long NumberToWrite;
```

Database/Update, Database/Physical
../liblqtext/wpblock.c

Adds the given Word Places to the database for the given WID. This routine is fairly low-level, and is made available in the API for efficiency. You should not attempt to use it without looking at examples in the lq-text clients that update the database, and also reading the source of the function itself. ❦ Returns: The number of places written.

---

API int
**LQT_DeleteWordFromIndex**(db, Word)

```
  t_LQTEXT_Database *db;
  char *Word;
```

Database/Update, Database/Words
../liblqtext/wordinfo.c

Deletes the given word and associated data from the database. The WID index entry for the LQT_WIDToWord function entry is retained, as is the widindex file record, with a match count of zero. If the word should appear in some subsequently indexed file, this space is reclaimed. ❦ Returns: 1. zero on success 2. −1 on error ❦ Notes: See LQC_UnIndexFile in the lqunindex client for an example of using this function.

Database/Update,
Database/Words
../liblqtext/pbcache.c

API void
**LQT_DeleteWordPlaces**(db, FirstBlock, WID)
```
  t_LQTEXT_Database *db;
  unsigned long FirstBlock;
  t_WID WID;
```
Deletes the word places from disk for a given WID, marking the corresponding data blocks as unused.<P> ¶ The given FirstBlock argument is the first block in the chain of the linked list of blocks for the given WID. If the data is contained entirely in the WID index block, LQT_DeleteWordPlaces should not be called, and this is a fatal error. ¶ LQT_DeleteWordPlaces does not remove the WID ⇔ Word mapping from the wordlist Key Value Database, and does not zero out the information in the widindex block. ❦ Errors: Fatal (E_BUG) error if FirstBlock or WID are zero. ❦ See Also: LQT_Deletepblock‡ (p. 55).

---

Database/Update,
Database/Files
../liblqtext/lastblk.c

API unsigned char *
**LQT_LastBlockInChain**(db, WID, Offsetp, FirstUnusedBytepp, BlockLengthp)
```
  t_LQTEXT_Database *db;
  t_WID WID;
  unsigned long *Offsetp; /* in: first offset; Out: last offset */
  unsigned char **FirstUnusedBytepp; /* out only */
  unsigned int *BlockLengthp;
```
Returns the last block in the chain for a given WID. The value may have been set previously by LQT_SetLastBlockInChain, or can be deduced by reading the chain from disk a block at a time until the end is reached. ❦ Returns: A pointer to the (extended) block in the data cache ❦ Errors: Fatal error (E_BUG) if the value cannot be determined ❦ See Also: LQT_SetLastBlockInChain‡ (p. 53).

---

Database/Update,
Database/Documents
../liblqtext/mkfinfo.c

API t_FileInfo *
**LQT_MakeFileInfo**(db, FileName)
```
  t_LQTEXT_Database *db;
  char *FileName;
```
Creates a t_FileInfo structure to describe the given FileName. This routine should only be used if you are going to add the given FileName to the given lq-text Database db; to get a FileInfo describing a file already in the index, use LQT_NameToFID and LQT_FIDToFileInfo. ❦ Returns: If the file is not already in the database, a new FID is allocated, and a newly malloc'd t_FileInfo object is returned, complete with a stdio FILE pointer already opened, either as a file or as a pipe, depending on the file type and filter table; it is the caller's responsibility to call LQT_DestroyFileInfo to free the memory and close the stdio stream. ¶ On error, or if the file is already in the database and has not changed since it was last indexed, a warning is issued and a NULL pointer is returned. ❦ Errors: If the file can't be found, or can't be opened, a warning is produced. ❦ See Also: LQT_DestroyFileInfo‡ (p. 35); LQT_NameToFID‡ (p. 14); LQT_FIDToFileInfo‡ (p. 9); LQT_GetFilter-

**LQT_Deletepblock** Database/Words, Database/Update, p. 55
**LQT_DestroyFileInfo** Database/Documents, Memory, p. 35
**LQT_FIDToFileInfo** Database/Retrieval, Database/Documents, p. 9
**LQT_NameToFID** Database/Retrieval, Database/Documents, p. 14
**LQT_SetLastBlockInChain** Database/Update, Database/Files, p. 53

TYPE‡ (p. 36); LQT_MAKEINPUT‡ (below).

---

API FILE *
**LQT_MakeInput**(db, FileInfo)
  t_LQTEXT_Database *db;
  t_FileInfo *FileInfo;

Database/Update, Database/Documents ../liblqtext/filters.c

Opens the document referred to by the given FileInfo for reading, using external input filters if necessary.</P>. ¶ The returned ſtdio ſtream may refer to a pipe or to a file; use LQT_DESTROYFILEINFO to close it. ¶ You muſt use LQT_DESTROYFILEINFO to close the file and free the memory ❦ Returns: A ſtdio ſtream open for reading, or NULL on error. ❦ Errors: Issues an error if a required external filter could not be ſtarted. ❦ See Also: LQT_MAKEFILEINFO‡ (opposite); LQT_DESTROYFILEINFO‡ (p. 35); LQT_GETFILTERTYPE‡ (p. 36).

---

API unsigned long
**LQT_MakeWordInfoBlock**(db, WordInfo, pblock)
  t_LQTEXT_Database *db;
  t_WordInfo *WordInfo;
  t_pblock *pblock;

Database/Update, Database/Words ../liblqtext/wordinfo.c

Tries to put the given pblock into the given WordInfo's index block, a buffer reserved for this purpose. ❦ Returns: 1. the number of places successfully added 2. 0 if no word places were given in pblock ❦ See Also: LQT_PUTWORDINFOINTOINDEX‡ (overleaf); LQT_MAKEWORDINFOBLOCKHEADER‡ (below). ❦ Errors: Warns if WordInfo already has a non-zero Offset.

---

LIBRARY void
**LQT_MakeWordInfoBlockHeader**(db, WordInfo, pblock)
  t_LQTEXT_Database *db;
  t_WordInfo *WordInfo;
  t_pblock *pblock;

Database/Update, Database/Words ../liblqtext/wordinfo.c

Writes a database header block (a WIDindex entry) into the given WordInfo. This is split into a separate routine so that the library can write a word block header tentatively, using a different format for the header if the header and the data all fit into the index block. LQT_MAKEWORDINFOBLOCKHEADER determines the format to use by whether WordInfo→Offset is non-zero. The difference is whether a fixed four bytes are used for the total number of word places for this word, or whether a variable number of bytes, using LQT_sWRITENUMBER, are written. In the latter case, update in place is not possible, and this format is therefore only used when WordInfo→Offset is zero, and any update would in any case have to read and rewrite the word index block.

**LQT_DestroyFileInfo** Database/Documents, Memory, p. 35
**LQT_GetFilterType** Database/Documents, p. 36
**LQT_MakeFileInfo** Database/Update, Database/Documents, p. 50
**LQT_MakeInput** Database/Update, Database/Documents, p. 51
**LQT_MakeWordInfoBlockHeader** Database/Update, Database/Words, p. 51
**LQT_PutWordInfoIntoIndex** Database/Update, Database/Words, p. 52

Database/Update, Database/Words ../liblqtext/wordinfo.c

API int
**LQT_PutWordInfoIntoIndex**(db, theWordInfo, Offset)

```
  t_LQTEXT_Database *db;
  t_WordInfo *theWordInfo;
  unsigned long Offset;
```

Each WordInfo ſtructure contains a pointer to a single data block, which is used to ſtore the widindex header. This speeds up indexing, since the header is needed at both the ſtart of writing out WordPlaces and at the end. LQT_PutWordInfoIntoIndex arranges that index block be written out to the widindex index file, using LQT_WriteWordInfoIndexBlock. ¶ A WID muſt have been allocated for this word with LQT_WriteWordAndWID for this word already, on this or some other program run. ¶ This routine is generally called after LQT_Writepblock. ❦ Returns: zero ❦ Errors: Warns if the WordInfo has a datablock but no offset. If ASCIITRACE was defined when the library was compiled, and if the LQTRACE_READAFTERWRITE trace flag is set, LQT_PutWordInfoIntoIndex checks that theWordinfo→WID corresponds to theWordInfo→Word, using LQT_WordToWID, and produces a fatal (E_BUG) error if not.

---

Database/Update, Database/Documents ../liblqtext/fileinfo.c

API int
**LQT_RemoveFileInfoFromIndex**(db, FileInfo)

```
  t_LQTEXT_Database *db;
  t_FileInfo *FileInfo;
```

Removes the given FileInfo from the FID⇔FileInfo maps. It is the caller's responsibility to ensure that the given FID is not referenced anywhere in a saved WordPlace. ❦ Returns: 1. zero on success 2. −1 on error ❦ See Also: LQT_NameToFID‡ (p. 14); LQT_DestroyFileInfo‡ (p. 35). ❦ Errors: Warns if the database can't be opened

---

Database/Update, Database/Documents ../liblqtext/fileinfo.c

API int
**LQT_RenameFileInIndex**(db, OldName, NewName)

```
  t_LQTEXT_Database *db;
  char *OldName;
  char *NewName;
```

Changes the filename associated with a FID, by finding the FID for the old filename and then replacing its filename. ❦ Returns: 1. zero on success 2. −1 on error ❦ Errors: Warns if the database can't be opened or the file isn't indexed.

---

Database/Update, Database/Documents ../liblqtext/fileinfo.c

API int
**LQT_SaveFileInfo**(db, FileInfo)

```
  t_LQTEXT_Database *db;
  t_FileInfo *FileInfo;
```

Stores the given t_FileInfo ſtructure in the database referred to by the given db argument, whence it can be retrieved by FID or by filename. ❦ Returns: 1. zero on success 2. −1 if error ❦ Errors: Warns if the database can't be opened or written to. ❦ See Also: LQT_RemoveFileInfoFromIndex‡ (above); LQT_DestroyFileInfo‡ (p. 35).

**LQT_DestroyFileInfo** Database/Documents, Memory, p. 35
**LQT_NameToFID** Database/Retrieval, Database/Documents, p. 14
**LQT_RemoveFileInfoFromIndex** Database/Update, Database/Documents, p. 52

API void
**LQT_SetLastBlockInChain**(db, WID, Offsetp, FirstUnusedBytep, theBlock)
  t_LQTEXT_Database *db;
  t_WID WID;
  unsigned long *Offsetp; /* In: last offset */
  unsigned char *FirstUnusedBytep;
  unsigned char *theBlock;

Database/Update, Database/Files ../liblqtext/lastblk.c

LQT_SetLastBlockInChain maintains the chainend file in the database directory; this contains the block number of the last block in the chain used to store data for a given WID. This allows lqaddfile to update an entry efficiently, as otherwise it has to read the entire chain from the start to determine the last block before it can start appending to it. Failing to call this function after changing the last block number for a given WID will result in a corrupt database. ¶ The given Offsetp is a pointer to a long, although the value is not changed; this is simply for consistency with other routines, and may change in the future. The FirstUnusedBytepp is currently used only for debugging; the value is recomputed from the data when it is used. ❦ Errors: Fatal error if the cache file can't be created, if it isn't already open. ❦ See Also: LQTp_FlushLastBlockCache‡ (overleaf); LQT_LastBlockInChain‡ (p. 50).

API void
**LQT_SortWordPlaces**(db, NumberOfWordPlaces, WordPlaces)
  t_LQTEXT_Database *db;
  unsigned long NumberOfWordPlaces;
  t_WordPlace *WordPlaces;

Database/Update, Database/Words ../liblqtext/lqsort.c

Sorts the given WordPlace array using Quicker Sort to the in-memory stop list, to be ignored by LQT_ReadWord. A WordPlace array must be sorted in ascending order by FID, then by Block In File, then by Word Within Block, in order to be written to the database. Since this is exactly the order generated by reading files one at a time from beginning to end, this routine is not currently used. ❦ Notes: Buggy, I think.

API void
**LQT_UpdateWIDMatchCount**(db, WID, AddedThese)
  t_LQTEXT_Database *db;
  t_WID WID;
  unsigned long AddedThese;

Database/Update, Database/Words ../liblqtext/wordinfo.c

Revises the count of the number of occurrences of the given word held in the WIDindex file. It is the caller's responsibility to ensure that this number is the same as the number of matches that are stored with LQT_WriteWordPlaces before the next call to LQT_GetWordPlaces. In particular, reducing the number of occurrences with this call will not cause word places to be deleted; a fatal (E_BUG) error will generally be produced on trying to read back a word with an inconsistent Match Count. ❦ Errors: It's a fatal error (E_BUG) if the WID isn't in

**LQT_LastBlockInChain** Database/Update, Database/Files, p. 50
**LQTp_FlushLastBlockCache** Database/Update, Database/Files, p. 54

the index.

---

Database/Update,
Database/Words
../liblqtext/wordinfo.c

```
API t_WID
LQT_WriteWordAndWID(db, Word, Length, WID)
  t_LQTEXT_Database *db;
  char *Word;
  int Length;
  t_WID WID;
```

Saves the WID → Word mapping in the wordlist database. ❦ Returns: the given WID. ❦ Errors: Fatal error if the database can't be opened, or if the word couldn't be stored. ❦ Notes: The reverse map, Word → WID, is performed using LQT_WIDToWord, and uses the copy of the word stored in the widindex block header. ❦ See Also: LQT_WIDToWord‡ (p. 15); LQT_WordToWID‡ (p. 16); LQT_PutWordInfoIntoIndex‡ (p. 52).

---

Database/Update,
Database/Physical
../liblqtext/wpblock.c

```
API unsigned long
LQT_WriteWordPlaces(
  db,
  WordPlaces,
  WID,
  LastStart,
  Block, DataStart, BlockLength,
  NextOffset, NextSize,
  NumberToWrite )
  t_LQTEXT_Database *db;
  t_WordPlace *WordPlaces;
  t_WID WID;
  unsigned long LastStart;
  unsigned char *Block;
  unsigned char *DataStart;
  unsigned int BlockLength;
  unsigned long NextOffset;
  unsigned long NextSize;
  unsigned long NumberToWrite;
```

Writes the given WordPlaces to disk. ¶ The given LastStart argument should be zero if the given Block pointer refers to data that is not to be stored in the overflow file ('data'). This will be the case when the first few matches are to be written into the widindex entry. If the LastStart argument is non-zero, it is the block number that will be passed as an argument to LQT_WriteBlock to save the block when it is full. ¶ The given NextOffset can either be zero or it can be the block offset in the data overflow file of a block that has been allocated using LQT_-FindFreeBlock; in the latter case, the NextLength argument is also passed on to LQT_-WriteWordPlaces. ❦ Returns: 1. the number of words added on success; 2. −1 if the file couldn't be opened. ❦ Notes: This routine is fairly low-level, and is made available in the API for efficiency. You should not attempt to use it without looking at examples in the lq-text clients that update the database, and also reading the source of the function itself. ❦ Errors: Warns if the file can't be opened.

---

Database/Update,
Database/Physical
../liblqtext/wpblock.c

```
API unsigned long
LQT_Writepblock(db, WordInfo, pblock)
  t_LQTEXT_Database *db;
  t_WordInfo *WordInfo;
  t_pblock *pblock;
```

Write out an entire (presumably new) data entry, and return a disk pointer to the start of the chain. ❦ Returns: the byte offset of the first block in the newly created chain ❦ Errors: Fatal (E_BUG) error on format or consistency check, etc.

---

Database/Update,
Database/Files
../liblqtext/lastblk.c

```
LIBRARY int
LQTp_FlushLastBlockCache(db)
  t_LQTEXT_Database *db;
```

Ensures that all entries in the last block cache are written out to disk. This routine must be called before a routine that has updated the database exits. ¶ This routine is registered as an action to be performed on a database close or sync, and so is called automatically by LQT_CloseDatabase and LQT_SyncDatabase; the ignored argument and the return value are for compatibility with LQT_AddActionOnClose. ❦ Errors: Warns if there are system problems writing the data or closing the associated file. ❦ See Also: LQT_SetLastBlockInChain‡ (previous page); LQT_LastBlockInChain‡ (p. 50); LQT_AddActionOnClose‡ (p. 3);
LQT_CloseDatabase‡ (p. 4)

## *Database/Words*

*Functions in this category are used to manipulate and update the vocabulary part of an lq-text index, and also deal with the low-level binary representation of lists of matches.*

*The pblock structure (referred to chiefly in this section) is defined in h/pblock.h; it is an in-memory representation of the data stored in one or more physical database blocks for a given word.*

*The t_WID type is a Word IDentifier: each distinct word in the vocabulary is assigned a unique number (a C unsigned long, starting at one rather than zero). This number is used as an index into a fixed-record-size file, 'widindex'. The record contained there stores the first few matches for the word, and possibly a pointer into the overflow file, 'data', where the rest of the matches are stored.*

---

API void
**LQT_Deletepblock**(db, pblock)
  t_LQTEXT_Database *db;
  t_pblock *pblock;

Deletes the word places for a given pblock, using LQT_DeleteWordPlaces. ¶ Like LQT_-DeleteWordPlaces, LQT_Deletepblock does not remove the WID ⇔ Word mapping from the wordlist Key Value Database, and does not zero out the information in the widindex block. ¶ In other words, the word is not removed from the database vocabulary, and subsequent calls to LQT_WordToWID will return the same WID value as before the call to LQT_Deletepblock.<P> ❦ See Also: LQT_Getpblock‡ (p. 13); LQT_DestroyWordInfo‡ (p. 63).

Database/Words, Database/Update
../liblqtext/pbcache.c

---

API t_WID
**LQT_GetMaxWID**(db)
  t_LQTEXT_Database *db;

Returns the largest currently allocated WID.

Database/Words
../liblqtext/getwid.c

Database/Words
../liblqtext/setwid.c

API int
**LQT_WriteCurrentMaxWID**(db)
  t_LQTEXT_Database *db;

Writes the value of the largeſt allocated WID to disk. This value is cached for efficiency, so LQT_WriteCurrentMaxWID muſt be called after allocating a new WID and before the program exits. ¶ Since LQT_WriteCurrentMaxWID is regiſtered as an action to be performed on closing or flushing a database, it will be called automatically by a call to either LQT_Close or LQT_Sync. ¶ The ignored argument is required by LQT_AddActionOnClose. ❦ See Also: LQT_AddActionOnClose‡ (p. 3); LQT_CloseDatabase‡ (p. 4); LQT_SyncDatabase‡ (p. 5).

Database/Words
../liblqtext/wordinfo.c

API void
**LQT_fprintWordInfo**(db, ſtream, W, Caller)
  t_LQTEXT_Database *db;
  FILE *ſtream;
  t_WordInfo *W;
  char *Caller;

Prints an ASCII representation of the given WordInfo pointer to the given ſtdio ſtream. The Caller argument is printed before each line of output, and is usually the name of the function calling LQT_fprintWordInfo.

**LQT_AddActionOnClose** Database/Database, p. 3
**LQT_CloseDatabase** Database/Database, p. 4
**LQT_SyncDatabase** Database/Database, p. 5

*The idea of ſtemming is that you put* Apple *and* Apples *under the same heading in the index; that is, collating them together, or conflating them.*

*Currently, the lq-text ſtemmer handles only plurals and possessives; a better one would also underſtand that run and ran and running go together, for example.*

*The more ſtemming you do, the lower the precision of matches, but the higher the recall. Since lq-text was originally designed for very high precision, ſtemming has not been a high priority.*

*Note that when matches are written to the database, the fact that ſtemming was applied is also recorded, using two bits (one for plurals and one for possessives) so that a query for* Apples *doesn't by default match* Apple *in the database, but a query for* Apple *will match both.*

*The principle is that the package should not infer more precision than was used in the query, but where higher precision was used, should take advantage of it where it can.*

---

Language/Stemming
../liblqtext/wordrule.c

API int
**LQT_EndsWord**(db, ch)
  t_LQTEXT_Database *db;
  int ch;

Returns non-zero only if the given character ch can appear within or at the end of a word. This function is normally a macro declared in the header file `<wordrules.h>` but can also be defined as a C function is greater complexity is needed and the indexing speed loss is not a concern. ❦ Returns: zero or non-zero. ❦ Bugs: This routine is only sensible for English. ❦ See Also: LQT_StartsWord‡ (p. 59); LQT_OnlyWithinWord‡ (overleaf).

---

Language/Stemming
../liblqtext/root.c

API char *
**LQT_GenerateWordFromRoot**(db, WordInfo, Flags)

t_LQTEXT_Database *db;
t_WordInfo *WordInfo;
unsigned int Flags;

LQT_GenerateWordFromRoot tries to generate the original word from the given flags. Sometimes multiple plurals reduce to the same singular, such as brothers and brethren both being forms of brother, and in these cases the generated word may be incorrect. Other cases include words ending in the letter o, which may or may not have has an es stripped off, so that SunOS (the operating system) is indexed as Suno, and incorrectly pluralised as Sunoes. ❦ Returns: A pointer to a static buffer ❦ Bugs: Should allow per-database stemming options. ❦ See Also: LQT_ReduceWordToRoot‡ (below); LQT_WordToWID‡ (p. 16); LQT_WIDToWord‡ (p. 15).

---

Language/Stemming
../liblqtext/wordrule.c

API int
**LQT_OnlyWithinWord**(db, ch)
t_LQTEXT_Database *db;
int ch;

Returns non-zero only if the given character ch can appear within a word but not at the start or end, and not repeated consecutively. For English, an apostrophe (') is normally considered to be the only such character; it's found in wouldn't, can't, and o'clock. You could also include the hyphen if you wanted, but it turns out to be best to index 'match-box' as two separate words with punctuation between them, rather than as a single word. ¶ This function is normally a macro declared in the header file `<wordrules.h>` but can also be defined as a C function is greater complexity is needed and the indexing speed loss is not a concern. ❦ Returns: zero or non-zero. ❦ Bugs: This routine is only sensible for English. ❦ See Also: LQT_StartsWord‡ (opposite).

---

Language/Stemming
../liblqtext/root.c

API char *
**LQT_ReduceWordToRoot**(db, WordInfo)
t_LQTEXT_Database *db;
t_WordInfo *WordInfo;

Reduces the word in the WordInfo pointed to by its argument to an English root, by stripping plurals and possessives. WordInfo→Length is modified as necessary, and WordInfo→Flags are updated by or'ing any necessary items from `<wordrules.h>`. The word can grow by up to two characters in length. It is the caller's responsibility to allocate enough space. You can also use the wordroot macro from `<wordrules.h>`, which calls LQT_ReduceWordToRoot only if it might make a change. ❦ Returns: A pointer to WordInfo's Word ❦ Bugs: This routine is only sensible for English. ❦ See Also: LQT_ReadWordFromStringPointer‡ (p. 15).

**LQT_ReadWordFromStringPointer** Database/Retrieval, Database/Documents, p. 15
**LQT_ReduceWordToRoot** Language/Stemming, p. 58
**LQT_StartsWord** Language/Stemming, p. 59
**LQT_WIDToWord** Database/Retrieval, Database/Words, p. 15
**LQT_WordToWID** Database/Retrieval, Database/Words, p. 16

API int
**LQT_StartsWord**(db, ch)
  t_LQTEXT_Database *db;
  int ch;

Language/Stemming
../liblqtext/wordrule.c

Returns non-zero only if the given character ch can appear at the start of a word. This function is normally a macro declared in the header file `<wordrules.h>` but can also be defined as a C function is greater complexity is needed and the indexing speed loss is not a concern. ❦ Returns: zero or non-zero. ❦ Bugs: This routine is only sensible for English. ❦ See Also: LQT_EndsWord‡ (p. 57); LQT_OnlyWithinWord‡ (opposite).

## *Language/Stop List*

*It's common practice in text retrieval to omit words from the database if they occur very often. For example, 'and', 'the' and 'to' don't seem to add very much information. However, in certain circumstances, such as 'The Times', or 'Bitwise and', the words are suddenly of great significance.*

*There are three approaches to this.*

*First, you can say that people looking for The Times are out of luck.*

*Second, you can index all of the words, and take a penalty on index size. This penalty is usually from one to thirty percent of the total index size, and is usually acceptable.*

*Thirdly, you could specify a list of contexts in which words in the stoplist are to be indexed anyway. There are three problems with this last approach. Firstly, you don't have enough context in a query to determine what to do about those words. Secondly, you have to think of all the contexts in advance; if you didn't think of 'the Times', the user would still be out of luck. Finally, lq-text doesn't support this third approach directly, although you could modify lq-text, perhaps using the routines in this category.*

---

```
ARGSUSED2*/
API void
LQT_InsertCommonWord(db, Root)
  t_LQTEXT_Database *db;
  char *Root;
```

Language/Stop List
../liblqtext/common.c

The given word will be ignored by LQT_READWORD. Note that if you ignore different words on retrieval than on indexing, lq-text will not be able to locate the exact text of matches, and phrase matching may have unexpected results. You should therefore not modify the stoplist once you have created an index. ❦ Bugs: The common list is shared by all lq-text databases. There is no way to remove a word from the stoplist. ❦ See Also: LQT_READSTOPLIST‡ (overleaf); LQT_WORDISINSTOPLIST‡ (overleaf).

Language/Stop Liﬆ
../liblqtext/common.c

API int
**LQT_ReadStopList**(db, CommonFile)
  t_LQTEXT_Database *db;
  CONST char *CommonFile;

Reads the named file, and adds any words found in it to the in-memory ﬆop liﬆ, to be ignored by LQT_-ReadWord. ❦ Returns: 1. the number of words added on success; 2. −1 if the file couldn't be opened. ❦ Errors: Warns if the file can't be opened. ❦ See Also: LQT_WordIsInStopList‡ (below). ❦ Bugs: There is no way to clear the ﬆop liﬆ; you can only add to it. The current implementation is inefficient if there are more than ten or so words. ❦ Notes: A future release may support a 'go liﬆ' of phrases every word of which is to be indexed.

Language/Stop Liﬆ
../liblqtext/common.c

API int
**LQT_WordIsInStopList**(db, WordInfo)
  t_LQTEXT_Database *db;
  t_WordInfo *WordInfo;

Returns 1 if the given word is in the ﬆop liﬆ, 0 otherwise. This function is called by the LQT_-ReadWord routines on each input word to determine whether to return it. ❦ Returns: 1. 1 if the word is in the ﬆop liﬆ 2. 0 otherwise ❦ Bugs: FirﬆCharBitMap is shared across all databases. You cannot have more than one database open at a time anyway at the moment, so this is not yet an issue...

*Routines in this category deal with allocating and freeing memory. The routines* EMALLOC‡ *(p. 68);* EREALLOC‡ *(p. 68);* ECALLOC‡ *(p. 67); and* EFREE‡ *(p. 67); routines are due to change shortly in a move to a slab allocation policy.*

---

API void
**LQT_DestroyWordInfo**(db, WP)
  t_LQTEXT_Database *db;
  t_WordInfo *WP;

Memory, Database/Words
../liblqtext/wordinfo.c

Deletes the given ſtructure from memory, reclaiming ſtorage. This routine does not affect the database. ❦ See Also: LQT_DESTROYFILEINFO‡ (p. 35); LQT_DELETEWORDFROMINDEX‡ (p. 49); LQT_MAKEWORDINFO‡ (below).

---

API t_WordInfo *
**LQT_MakeWordInfo**(db, WID, Length, Word)
  t_LQTEXT_Database *db;
  t_WID WID;
  int Length;
  unsigned char *Word;

Memory, Database/Words
../liblqtext/wordinfo.c

Conſtructs a new t_WordInfo ſtructure containing a malloc'd and NUL terminated copy of the given word. The word as passed into LQT_-MAKEWORDINFO need not be NUL terminated; the Length parameter is the number of bytes in the Word ſtring, not counting the trailing NUL, if present. ❦ See Also: LQT_READWORDFROMSTRINGPOINTER‡ (p. 15); LQT_DESTROYWORDINFO‡ (above); LQT_WORDTOWID‡ (p. 16). ❦ Errors: Fatal error if there isn't enough memory

# *Other/Terminal*

*This category contains any remaining holdovers from the libcurses era. These functions are being moved out of the main library and into the clients; it's possible that a new library will be created to create them in time.*

---

API void
**LQU_CursesSafeystem**(string, retvalp)
  char *string;
  int *retvalp;

runs the given string as a system command, using system(3); the terminal modes are restored before and after. ❦ Restrictions: This routine should not be used and will be deleted from the next release; it is only useful for curses-based clients, and should be provided separately.

Other/Terminal
../liblqtext/system.c

*The functions in this category provide wrappers around the system-provided malloc, free and friends. The reasons for using these functions are as follows:*

*To provide consistant error messages;*

*to aid in porting;*

*To aid in debugging.*

*If the compile-time manifest* MALLOCTRACE *is defined (for example, with* –DMALLOCTRACE=1 *as a compiler option), these routines provide tracing output to standard error which can be used to detect memory leaks.*

---

char *
**ecalloc**(What, Number, Size)
  CONST char *What;
  unsigned int Number;
  unsigned int Size;

Utilities/Memory
../liblqutil/malloc.c

Allocates sufficient memory to hold the given Number of objects of the given Size, after taking alignment constraints into account; the system-supplied calloc function is used. ¶ If there is not enough memory, a fatal error is generated. The What argument is included in any such error message, and should be a human-readable description of the error, as an aid to help the user understand exactly what failed. ¶ A future release of lq-text will have an improved memory allocation interface. ❦ Errors: A fatal (E_FATAL | E_MEMORY) error is produced if memory is exhausted. ❦ See Also: EMALLOC‡ (overleaf); EFREE‡ (below); ERROR‡ (p. 23).

---

void
**efree**(String)
  char *String;

Utilities/Memory
../liblqutil/malloc.c

Returns the memory used by an object to the system, using the system-provided free function. ¶ A future release of lq-text will have an improved memory allocation interface. ❦ Errors: A warning (E_-WARN) is produced a NULL pointer is passed as an argument.

**Error** Error Handling, p. 23
**efree** Utilities/Memory, p. 67
**emalloc** Utilities/Memory, p. 68

Utilities/Memory
../liblqutil/malloc.c

```
char *
emalloc(What, nbytes)
  CONST char *What;
  unsigned nbytes;
```

Allocates the given number of bytes of memory and returns a pointer to it, using the system-supplied malloc function. ¶ If there is not enough memory, a fatal error is generated. The What argument is included in any such error message, and should be a human-readable description of the error, as an aid to help the user understand exactly what failed. ¶ A future release of lq-text will have an improved memory allocation interface. ❦ Errors: A fatal (E_FATAL | E_MEMORY) error is produced if memory is exhausted. ❦ See Also: ECALLOC‡ (previous page); EFREE‡ (previous page); ERROR‡ (p. 23).

Utilities/Memory
../liblqutil/malloc.c

```
char *
erealloc(Object, NewSize)
  char *Object;
  unsigned int NewSize;
```

Changes the size of the given Object, either by extending the area of memory allocated to it or by allocating a new area, copying the data and freeing the original storage area. ¶ If insufficient memory is available, a fatal (E_FATAL) error is produced, which includes the given What argument as a textual (human-readable) description of the object. ¶ The system-supplied realloc function is used. ❦ Returns: A pointer to the newly sized object; in most implementations this will almost always be a new copy of the object. A future release of lq-text will have an improved memory allocation interface. ❦ Errors: A fatal (E_FATAL | E_MEMORY) error is produced if memory is exhausted. ❦ See Also: EMALLOC‡ (above); EFREE‡ (previous page); ERROR‡ (p. 23).

**Error** Error Handling, p. 23
**ecalloc** Utilities/Memory, p. 67
**efree** Utilities/Memory, p. 67
**emalloc** Utilities/Memory, p. 68

## *Utilities/System*

*Interactions with the operating environment, such as fetching a user's login directory, are listed here. In general, lq-text has minimal involvement with the operating system apart from memory allocation and the file system, so there is not much in this category.*

---

API char *
**LQU_GetLoginDirectory**()

Utilities/System
../liblqutil/homedir.c

Determines the home directory of the current user. It returns the value of the environment variable $HOME if it is set. If this isn't set, or is empty, or does not point to a directory, the password file (or Yellow pages) is consulted instead. ❦ Returns: The directory name in a malloc's string; it is the caller's responsibility to free this string if it should no longer be needed. If the home directory cannot be determined, a NULL pointer is returned; this might happen if the user's entry in /etc/passwd was removed while the program was running, or if the Yellow Pages (NIS) service became unavailable.

*The routines in this category are generally wrappers around Unix system or library calls, or are useful routines for file handling.*

*The wrapper routines exist so that helpful and precise error messages can be generated in failure cases.*

*The other routines are for items such as determining whether a filename refers to a file or a directory, reading a file into memory a line at a time, or determining whether a file is empty.*

*As with all of the Utilities categories, none of these routines are specific to the lq-text database in any way, but they are all used by lq-text.*

---

API off_t
**LQU_Elseek**(Severity, Name, What, fd, Position, Whence)

Utilities/Files
../liblqutil/elseek.c

```
  int Severity;
  CONST char *Name;
  CONST char *What;
  int fd;
  long Position;
  int Whence;
```

This is a wrapper for the lseek(2) system call. On an error, the given Name (which should reflect the corresponding file name, but need not be suitable to access that file) and What, which should be a terse description of the way in which the program is using the file, are used to construct a message passed to Error with the given Severity. ¶ The fd, Position and Whence arguments are as for the lseek(2) system call. ❦ Returns: The new file offset on success, or −1 on failure. ❦ Errors: Generates an Error at the given Severity if lseek fails, adding (with bitwise or) E_SYS if appropriate.

❦ Example:

```
  Where = LQU_lseek(E_FATAL, "passwd", "user database", 0, 0L, SEEK_SET);
```

Utilities/Files
../liblqutil/fEopen.c

API int
**LQU_Eopen**(Severity, Name, What, Flags, Modes)

```
  int Severity;
  CONST char *Name;
  CONST char *What;
  int Flags;
  int Modes;
```

Opens the named file with the given Flags and Modes, as per open(2). If the open fails, an error is generated with the given severity, and including both the file name (Name) and description (What). A diagnosis of the problem is also generated, using errno and examining the filename to determine if (for example) a component of the given path was not a directory. This generally produces much more specific, and hence, clearer, error messages than using perror(3) would give. ❦ Returns: a valid file descriptor on success, or −1 if the file couldn't be opened. If E_FATAL was given, LQU_Eopen does not return after an error.

❦ Example:

```
   *   LQU_Eopen(E_FATAL, "foo.c", "input C source", O_RDONLY, 0)
```

Utilities/Files
../liblqutil/eread.c

API int
**LQU_Eread**(Severity, Name, What, fd, Buffer, ByteCount)

```
  int Severity;
  CONST char *Name;
  CONST char *What;
  int fd;
  char *Buffer;
  int ByteCount;
```

This routine provides an error-checking wrapper around the read(2) system call. If the underlying read() returns −1, a diagnostic message is printed using by calling Error at the given Severity (bitwise or'd with E_SYS if appropriate). The message includes What, which should be a short, succinct summary of the purpose of the file, and Name, which is normally given as the name of the file, but could be any string. ❦ Returns: the number of bytes read on success, or −1 on an error. If E_FATAL was given, LQU_Eread does not return after an error.

❦ Example:

```
  nBytesRead = LQU_Eread(E_FATAL, "passwd", "list of users", 0, p, 12);
```

❦ Notes: There are several error flags, such as E_BUG, that include E_FATAL. See <error.h> for the current list. ¶ The example does not need to check to see whether nBytesRead is less than 0, since in that case the program would exit. LQU_Eread can, however, return a number other than ByteCount, just as the underlying system call read(2) can, and in the same circumstances. The caller of LQU_Eread should therefore check that the expected number of bytes were returned.

API int
**LQU_IsDir**(Dir)
  CONST char *Dir;

Utilities/Files
../liblqutil/isdir.c

returns 1 if and only if the given path is a directory. See the description for stat(2) for more details. ❦ Errors: A fatal error is issued if LQU_IsDir is called with a null string; a warning is issued if the string is of length zero.

---

API int
**LQU_IsFile**(Path)
  CONST char *Path;

Utilities/Files
../liblqutil/isfile.c

Determines whether the given Path refers to a regular file. Devices (such as /dev/null or a terminal), and directories in particular are not regular files. The Unix command ¶ find filename -type f -print ¶ will print out filename if and only if LQU_IsFile would return 1 for the same filename. ❦ Returns: 1. 1 if the given Path represents a regular file 2. zero otherwise ❦ Notes: There is tracing in here so that you can see which files are being investigated by the calling program; tracing is available if the liblqutil library was compiled with -DASCIITRACE; if so, you can set the FindFile trace flag (LQTRACE_FINDFILE) to see tracing for this routine. The -t "FindFile|Verbose" command-line option will do this. ¶ On systems that have the trace, strace or truss utility, investigate using that instead.

---

API int
**LQU_IsNonEmptyFile**(Path)
  CONST char *Path;

Utilities/Files
../liblqutil/isnzfile.c

Determines whether the given Path names a regular file that contains data. In other words, the file must have the its stat st_mode's S_IFMT field set to S_-IFREG, and must also have a non-zero st_size field; see the stat(2) man page. ❦ Returns: Non-zero if and only if Path names a regular file of non-zero length ❦ Notes: There is tracing in here so that you can see which files are being investigated by the calling program; tracing is available if the liblqutil library was compiled with -DASCIITRACE; if so, you can set the FindFile trace flag (LQTRACE_FINDFILE) to see tracing for this routine. The -t "FindFile|Verbose" command-line option will do this. ¶ On systems that have the trace, strace or truss utility, investigate using that instead. ❦ See Also: LQU_IsFile‡ (above); LQU_IsDir‡ (above).

---

API long
**LQU_ReadFile**(Severity, Name, What, Lines, Flags)
  int Severity;
  CONST char *Name;
  CONST char *What;
  char ***Lines;
  int Flags;

Utilities/Files
../liblqutil/readfile.c

Reads the file named by the Name argument, and returns a pointer to an array of pointers to the start of each line in the file. ¶ The Flags argument is any combination of flags from `<lqutil.h>` combined with bitwise or; in practice, however, LQUF_NORMAL is the most frequently used flag, which is a bitwise or of all of the flags described below.

¶ The flags are as follows: 1. LQUF_IGNBLANKS to throw away blank lines; 2. LQUF_-IGNSPACES to discard leading and trailing spaces; 3. LQUF_IGNHASH to discard leading comments (# with a hash-sign); 4. LQUF_IGNALLHASH to discard comments (# with a hash-sign); 5. LQUF_ESCAPEOK to accept \# and \\ as # and \. ¶ This is the file descriptor version of LQU_fReadFile. ¶ In the event of an error, the given Severity argument is passed to Error, along with the given What argument, which should be a brief English description, perhaps of the order of three words long, of the file. ❦ Returns: the number of lines read, if any. The char ** pointed to by the Lines argument is set to point to an array of ſtrings, each containing one line of text, NUL-terminated with trailing newlines removed. If E_-FATAL was given, LQU_fReadFile does not return after an error.

❦ Example:

```
int numberOfLines;
*   char **theLines;
int i;
*
*   numberOfLines = LQU_fReadFile(E_FATAL,
  "julian.txt",
  "Book of Meditations",
  &theLines,
  LQUF_NORMAL
);
*
for (i = 0; i < numberOfLines; i++) {
  printf("Line %d was: %s\n", i, Lines[i]);
  efree(Lines[i]);
}
efree((char *) Lines);
```

❦ Errors: Generates a Warning or Error of the given Severity if the file can't be opened, and attempts to diagnose the cause. ❦ See Also: LQU_fReadFile‡ (opposite).

---

Utilities/Files
../liblqutil/freadln.c

API char *
**LQU_StealReadLineBuffer**()

Returns the internal line buffer used by LQU_-fReadLine, and also causes LQU_fReadLine to allocate a new buffer the next time it is called. In this way, you can read lines with LQU_fReadLine, and save any that you are intereſted in keeping by calling LQU_StealReadLineBuffer, without having to copy the data. ¶ The buffer returned may be longer than necessary to contain the line that was laſt ſtored there by LQU_fReadLine by up to LQT_READLINE_SLOP bytes; use erealloc to shrink it if desired. The LQT_READLINE_SLOP conſtant is defined in freadln.c as 30 bytes. ❦ Returns: a pointer to the buffer, or NULL if there isn't one yet.

---

Utilities/Files
../liblqutil/fEopen.c

API void
**LQU_fEclose**(Severity, fp, Name, What)
  int Severity;
  FILE *fp;
  CONST char *Name;
  CONST char *What;

Closes the given file descriptor, printing error messages if necessary. ❦ Returns: There is no return value. If E_FATAL was given, LQU_fEclose does not return after an error.

---

Utilities/Files
../liblqutil/fEopen.c

API FILE *
**LQU_fEopen**(Severity, Name, What, Mode)
  int Severity;
  CONST char *Name;
  CONST char *What;
  CONST char *Mode;

This is the stdio equivalent of LQU_Eopen. ❦ Returns: a freshly opened file pointer (FILE *) on success, or NULL if the file couldn't be opened. If E_-FATAL was given, LQU_fEopen does not return after an error. ❦ Errors: Warns if the file can't be opened.

---

Utilities/Files
../liblqutil/freadf.c

API long
**LQU_fReadFile**(Severity, f, Name, What, Lines, Flags)
  int Severity;
  FILE *f;
  CONST char *Name;
  CONST char *What;
  char ***Lines;
  int Flags;

Reads the named file (Name), and mallocs an array of char * pointers to the start of each line read. The number of lines returned may be less than the number in the file, since by default LQU_fReadFile ignores blank or commented lines. Comments are denoted by a # as the first non-blank character on the line. If the file can't be opened, memory is exhausted, LQU_ReadFile() calls Error() with the given Severity, and with an error message constructed out of What, which should be a short (e.g. 3-word) description of the purpose of the file. The Flags argument can contain any of the following, combined with or (|): 1. UF_IGNBLANKS to throw away blank lines, 2. UF_IGNSPACES to discard leading and trailing spaces, 3. UF_IGNHASH to discard leading comments (# with a hash-sign) 4. UF_IGNALLHASH to discard comments (# with a hash-sign) 5. UF_ESCAPEOK to accept ¶ In addition, UF_NORMAL is defined to be UF_IGNBLANKS | UF_IGNSPACES | UF_IGNHASH | UF_ESCAPEOK and use of this in reading files is strongly encouraged to provide a consistent file format. ❦ Returns: 1. a pointer to the array of lines, in Lines 2. the number of lines allocated. 3. −1 if the file couldn't be opened. ❦ Errors: Warns (with the given severity | E_SYS) if the file can't be opened.

Utilities/Files
../liblqutil/freadln.c

```
API int
LQU_fReadLine(f, Linep, Flags)
  FILE *f;
  char **Linep;
  int Flags;
```

Reads the next input line from the given file into a static buffer. The buffer is allocated with malloc and resized dynamically, but is owned by LQU_fReadLine and should not be free'd or overwritten. ¶ The LQU_StealReadLineBuffer function can be used to obtain the buffer; LQU_fReadLine will allocate a new one the next time it is called. ¶ The given Flags are treated as for LQU_fReadFile, which currently calls this routine directly. Note that, as for LQU_fReadFile, blank lines are skipped if the corresponding flag is given. In this case, LQU_fReadLine will never return a pointer to a blank line, but will continue reading lines from the file until a non-blank one is found. ❦ Returns: a pointer to the line, in Line, and also the number of bytes in the line; –1 is returned on EOF, in which case the Line pointer should not be used.

*A Name Space is a set of ſtring-valued names that map into C variables. In other words, it's a symbol table.*

*The main use for these is in the lqkwic client, but they are deſtined for higher things, including internationalised message support and configuration options.*

*A new facility, the Glue Interpreter, will be available in the next release; this generalises the little language used by the lqkwic client, and provides something rather like printf and scanf but with named variables (and higher efficiency). if you are working in this area, or would like to know more, you should send mail to Liam Quin (lee@sq.com) and ask him for the ſtate of progress on Glue.*

API t_NameRef
**LQU_FirstNameRef**(NameSpace)
  t_NameSpace *NameSpace;

Utilities/Name Space
../liblqutil/namespace.c

Used in conjunction with LQU_NextNameRef to iterate over all of the Names in a Name Space. ❦ Returns: A reference to the firſt Name in the given Name Space, if there are any. Use LQU_NameRefIsValid() to determine if the returned reference is valid; if not, LQU_NameRefIsError will determine if there was an error, and LQU_GetNameError will handle the error using Error().
❦ Example:

```
 *          t_NameRef NameRef;
 *
 *          for (
 *            NameRef = LQU_FirstNameRef(NameSpace);
 *            LQU_NameRefIsValid(NameSpace, NameRef);
 *            NameRef = LQU_NextNameRef(NameSpace, NameRef)
 *          ){
 *            now use the Name Reference:
 *            printf("%s\n", LQU_GetNameFromNameRef(NameRef));
 *          }
```

❦ See Also: LQU_GetNameFromNameRef‡ (below); LQU_GetTypeFromNameRef‡ (below); LQU_NameRefIsValid‡ (opposite); LQU_NameRefIsError (undocumented);

---

Utilities/Name Space
../liblqutil/namespace.c

API char *
**LQU_GetDescriptionFromNameRef**(NameRef)
  t_NameRef NameRef;

Returns the textual description of the variable associated with a given NameRef, or NULL if there is none. ¶ Where the description is available, it is intended to be presented to the user, for example in error messages, and not to be parsed. ❦ Notes: The NameRef muﬅ be valid.

---

Utilities/Name Space
../liblqutil/namespace.c

API char *
**LQU_GetDescriptionFromNameSpace**(NameSpace)
  t_NameSpace *NameSpace;

Returns a pointer to the textual description of the given NameSpace. The text is in private memory, and so should not be freed by the caller.

---

Utilities/Name Space
../liblqutil/namespace.c

API char *
**LQU_GetNameFromNameRef**(NameRef)
  t_NameRef NameRef;

Retrieves the name of the given NameRef as a ﬅring. The NameRef muﬅ be valid. ❦ Returns: A pointer to the name; you should not free this ﬅring. ❦ See Also: LQU_NameRefIsValid‡ (opposite).

---

Utilities/Name Space
../liblqutil/namespace.c

API t_NameType
**LQU_GetTypeFromNameRef**(NameRef)
  t_NameRef NameRef;

Returns the type of the variable associated with the given NameRef. ¶ The types are defined in <namespace.h> as an enumerated type. ❦ Notes: The NameRef muﬅ be valid.

API void *
**LQU_GetVariableFromNameRef**(NameRef)

Utilities/Name Space
../liblqutil/namespace.c

Returns a pointer to the variable associated with a given NameRef. ¶ You have to caſt the result of this function, perhaps using LQU_GetTypeFromNameRef and a switch, since C lacks runtime type information. ❦ Notes: The NameRef muſt be valid.

---

API int
**LQU_NameRefFunctionTakesArgument**(NameRef)
t_NameRef NameRef;

Utilities/Name Space
../liblqutil/namespace.c

Returns non-zero if the function pointer associated with the given Name Ref is a pointer to a function that takes an argument. Before calling this function (or macro), you should check that LQU_NameRefVariablePointsToFunction returns non-zero for the given NameRef. ❦ Notes: The NameRef muſt be valid.

---

API int
**LQU_NameRefIsValid**(NameSpace, NameRef)
t_NameSpace *NameSpace;
t_NameRef NameRef;

Utilities/Name Space
../liblqutil/namespace.c

Determines whether the given NameRef is a valid reference to a name in the given NameSpace. ¶ A NameRef is invalid if it is a NULL pointer, or if the Name to which it refers has been deleted from the NameSpace. ❦ Notes: This function does <E>not</E> check to see whether a NameRef has been corrupted; the given NameRef muſt either be NULL, or have previously been a valid NameRef in the given NameSpace. ❦ Returns: Non-zero if the NameRef is valid, and zero otherwise. ❦ See Also: LQU_StringToNameRef‡ (p. 82); LQU_SetNameVariable‡ (p. 82).

---

API char *
**LQU_NameRefToString**(NameRef)
t_NameRef NameRef;

Utilities/Name Space
../liblqutil/namespace.c

Converts the value pointed to by the variable associated with the given Name Reference into a ſtring. ❦ Returns: a dynamically allocated ſtring, which the caller muſt free. ❦ See Also:
LQU_NameRefValueToString‡ (below); LQU_SetNameTypeAndVariable‡ (p. 81); LQU_GetNameFromNameRef‡ (opposite); LQU_GetVariableFromNameRef‡ (above).

---

API char *
**LQU_NameRefValueToString**(NameRef)
t_NameRef NameRef;

Utilities/Name Space
../liblqutil/namespace.c

Converts the value pointed to by the variable associated with the given Name Reference into a ſtring. ❦ Returns: a dynamically allocated ſtring, which the caller muſt free. ❦ See Also:

**LQU_GetNameFromNameRef** Utilities/Name Space, p. 78
**LQU_GetVariableFromNameRef** Utilities/Name Space, p. 79
**LQU_NameRefValueToString** Utilities/Name Space, p. 79
**LQU_SetNameTypeAndVariable** Utilities/Name Space, p. 81
**LQU_SetNameVariable** Utilities/Name Space, p. 82
**LQU_StringToNameRef** Utilities/Name Space, p. 82

LQU_SetNameTypeAndVariable‡ (p. 81); LQU_GetNameFromNameRef‡ (p. 78); LQU_GetVariableFromNameRef‡ (previous page).

---

Utilities/Name Space
../liblqutil/namespace.c

API int
**LQU_NameRefVariableAllocatedByLibrary**(NameRef)
t_NameRef NameRef;

Determines whether the variable associated with the given NameRef was allocated automatically (and is anonymous), or whether it was allocated externally and supplied to on of the Name Space creation functions. ❦ Notes: The NameRef must be valid.

---

Utilities/Name Space
../liblqutil/namespace.c

API int
**LQU_NameRefVariablePointsToFunction**(NameRef)
t_NameRef NameRef;

Returns non-zero if the variable associated with the given NameRef has previously been marked a pointer to a function, for example with LQU_SetNameRefVariablePointsToFunction. ❦ Notes: The NameRef must be valid.

---

Utilities/Name Space
../liblqutil/namespace.c

API t_NameSpace *
**LQU_NameSpaceTableToNameSpace**(Name, theTable)
char *Name;
t_NameSpaceTable theTable;

Converts a Name Space Table into a Name Space. This is useful if you have a statically initialised Name Space Table, for example. ¶ The new Name Space has the given Name as its name. The string is pointed to but not copied, and should therefore be allocated by the caller if it is not static data. The entries in the Name Space Table are copied, but their Name fields are simply pointed to. ❦ Returns: the newly created Name Space if successful. Currently, a failure is always fatal. ❦ See Also: LQU_StringToNameRef‡ (p. 82); LQU_SetNameVariable‡ (p. 82).

---

Utilities/Name Space
../liblqutil/namespace.c

API char *
**LQU_NameRefTypeToString**(NameType)
t_NameType NameType;

Returns a string representation of the given NameType. ❦ Returns: A statically allocated string, which need not be freed. ❦ See Also: LQU_SetNameTypeAndVariable‡ (opposite); LQU_GetNameFromNameRef‡ (p. 78).

**LQU_GetNameFromNameRef** Utilities/Name Space, p. 78
**LQU_GetVariableFromNameRef** Utilities/Name Space, p. 79
**LQU_SetNameTypeAndVariable** Utilities/Name Space, p. 81
**LQU_SetNameVariable** Utilities/Name Space, p. 82
**LQU_StringToNameRef** Utilities/Name Space, p. 82

API t_NameRef
**LQU_NextNameRef**(NameSpace, NameRef)
  t_NameSpace *NameSpace;
  t_NameRef NameRef;

Utilities/Name Space
../liblqutil/namespace.c

Used in conjunction with LQU_NextNameRef to iterate over all of the Names in a Name Space. ❦ Returns: A reference to the first Name in the given Name Space, if there are any. Use LQU_NameRefIsValid() to determine if the returned reference is valid; if not, LQU_NameRefIsError will determine if there was an error, and LQU_GetNameError will handle the error using Error(). ❦ See Also: LQU_GetNameFromNameRef‡ (p. 78); LQU_GetTypeFromNameRef‡ (p. 78); LQU_NameRefIsValid‡ (p. 79); LQU_NameRefIsError (undocumented);

---

API void
**LQU_SetNameRefFunctionTakesArgument**(NameRef)
  t_NameRef NameRef;

Utilities/Name Space
../liblqutil/namespace.c

Stores within the NameRef the fact that the variable associated with it is a pointer to a function that takes an argument. The NameRef must previously have been marked as being associated with a function pointer using LQU_SetNameRefVariablePointsToFunction. ❦ Notes: The NameRef must be valid.

---

API int
**LQU_SetNameRefVariableAllocatedByLibrary**(NameRef)
  t_NameRef NameRef;

Utilities/Name Space
../liblqutil/namespace.c

Stores within the NameRef the fact that the variable associated with it is a piece of dynamically allocated memory internal to the Name Space library. ❦ Notes: The NameRef must be valid. ¶ This function should not be used by client software.

---

API void
**LQU_SetNameRefVariablePointsToFunction**(NameRef)
  t_NameRef NameRef;

Utilities/Name Space
../liblqutil/namespace.c

Stores within the NameRef the fact that the variable associated with it is a pointer to a function. ❦ Notes: The NameRef must be valid.

---

API t_NameRef
**LQU_SetNameTypeAndVariable**(theNameRef, theNameType, theVariable)
  t_NameRef theNameRef;
  t_NameType theNameType;
  void *theVariable;

Utilities/Name Space
../liblqutil/namespace.c

Associates the given NameRef with the given Variable, first changing the remembered type of the NameRef. You should pass a pointer to the variable you want to use. The variable itself should be static if there is any chance of the Name within the NameSpace being used after the variable

**LQU_GetNameFromNameRef** Utilities/Name Space, p. 78
**LQU_GetTypeFromNameRef** Utilities/Name Space, p. 78
**LQU_NameRefIsValid** Utilities/Name Space, p. 79

has gone out of scope. ❦ Returns: The given NameRef, possibly changed, is returned.
❦ Example:

```
ſtatic int MyToes = 10;
LQU_SetNameTypeAndVariable(NameRef, LQU_NameType_Integer, &MyToes);
```

❦ See Also: LQU_SetNameVariable‡ (below).

---

Utilities/Name Space
../liblqutil/namespace.c

API t_NameRef
**LQU_SetNameValue**(NameRef, Value)
  t_NameRef NameRef;
  void *Value;

Sets the value of the variable associated with the given NameRef. ❦ Returns: the given NameRef. ❦ See Also: LQU_SetNameVariable‡ (below); LQU_SetNameTypeAndVariable‡ (previous page).

---

Utilities/Name Space
../liblqutil/namespace.c

API t_NameRef
**LQU_SetNameVariable**(NameRef, Variable)
  t_NameRef NameRef;
  void *Variable;

Associates a variable with a Name that you have retrieved from a Name Space. You should pass a pointer to the variable, which muſt remain in scope for as long as the Name can be accessed. ❦ Returns: the given Name Reference. ❦ See Also: LQU_SetNameTypeAndVariable‡ (previous page); LQU_StringToNameRef‡ (below).

---

Utilities/Name Space
../liblqutil/namespace.c

API t_NameRef
**LQU_StringToNameRef**(theNameSpace, theName)
  t_NameSpace *theNameSpace;
  char *theName;

Treats the given 'theName' ſtring as a Name, and looks this up in the given NameSpace. If the NameSpace allows neſted NameSpace references, the Name is allowed to have any number of prefixes consiſting of a name followed by a dot; the name muſt be the name of a NameSpace in the NameSpace being searched, and in this case the search proceeds using the newly found NameSpace on the reſt of the ſtring. ❦ Returns: the NameRef, or NULL
❦ Example:

```
If given the ſtring 'Children.Boys.Simon', and a NameSpace called
'People', LQU_StringToNameRef will search 'People' for a NameSpace
called Children, and if that should succeed, it will then search
'Children' for a NameSpace called 'Boys'.
If this laſt search succeeds, the namespace 'Boys' is searched
for 'Simon', and the result, either the NameRef called 'Simon' or
NULL for failure, is returned.
```

❦ See Also: LQU_SetNameTypeAndVariable‡ (previous page); LQU_GetVariableFromNameRef‡ (p. 79).

**LQU_GetVariableFromNameRef** Utilities/Name Space, p. 79
**LQU_SetNameTypeAndVariable** Utilities/Name Space, p. 81
**LQU_SetNameVariable** Utilities/Name Space, p. 82
**LQU_StringToNameRef** Utilities/Name Space, p. 82

## *Utilities/Numeric Range*

*This category provides some routines for maniplating an ASCII representation of numeric ranges, and corresponding in-memory data structures. It's useful for such things as lists of pages to print (5,12-20,37-), and is used by some of the lq-text clients to determine which matches to process.*

---

API int
**LQU_LargerThanRangeTop**(n, Range)
  CONST int n;
  CONST t_Range *Range;

Utilities/Numeric Range
../liblqutil/range.c

Use for efficiency, to determine whether a given number is larger than the largest value accepted by the given range. Passing a range that ended with a hyphen (for example, 1,2,5–7,12-) will always produce a zero result, even if n falls within a 'hole' in the range, as for 4, 8, 9, 10 and 11 in the example here. ❦ Returns: 1. 0 if the number is not entirely beyond the given range 2. 1 otherwise ❦ See Also: LQU_StringToRange‡ (below); LQU_NumberWithinRange‡ (below).

---

API int
**LQU_NumberWithinRange**(n, Range)
  CONST int n;
  CONST t_Range *Range;

Utilities/Numeric Range
../liblqutil/range.c

Determine whether a given number, n, falls within a given range. A range is a list like '–4,12–30,40,100-', to match ¶ 1, 2, 3, 4, 12, 13...29, 30, 40, 100, 101, 102, ... ¶ A space can be used instead of a comma. The range generates the range '–1,2-', matching all numbers ❦ Returns: 1. 1 if the n is within (matched by) the given range 2. 0 otherwise ❦ See Also: LQU_StringToRange‡ (below); LQU_LargerThanRangeTop‡ (above).

---

API t_Range *
**LQU_StringToRange**(String)
  CONST char *String;

Utilities/Numeric Range
../liblqutil/range.c

Converts the given string to a range; integers can subsequently be matched against the range with LQU_NumberWithinRange. ❦ Returns: 1. a pointer to a range on success 2. 0 otherwise ❦ Errors: A null string argument produces a fatal error. Syntax errors are also fatal. ❦ See Also: LQU_LargerThanRangeTop‡ (above); LQU_NumberWithinRange‡ (above).

*The routines in this category are for general ſtring handling; in addition,* `<globals.h>` *contains definitions for* STREQ *and* STRNCMP, *after an idea by* Henry *Spencer; these are not currently documented here.*

*The joinſtr routines are for joining two or three ſtrings together to make a single longer one; these are useful for conſtructing full pathnames out of a directory and a filename.*

---

API char *
**LQU_DownCase**(String)
  CONST char *String;

Utilities/Strings
../liblqutil/downcase.c

Returns a pointer to a ſtatic buffer containing a copy of the given ſtring in which all upper-case characters have been converted to lower case. The buffer grows automatically, and requires that the given String be nul-terminated. ❦ Notes: Relies on correct support from isupper, as described in ctype(3). On some syſtems, this function returns garbage if a character with the top bit set is teſted, and LOCALE has not been set. ❦ Bugs: The argument is not checked to see if it is a NULL pointer.

---

API char *
**LQU_ReverseString**(ſtart, end, type)
  char *ſtart;
  char *end;
  int type;

Utilities/Strings
../liblqutil/revſtr.c

Reverses the bytes in the given ſtring; if 'type' is even, the individual whitespace-delimited words are reversed in place; if type is even, he entire ſtring is reveresed. The process is repeated with (type - 1) until type is zero. Hence, a reverse type of zero does nothing, and a reverse type of one reverses the ſtring in place; a reverse type of 2 will reverse the order of the words in the ſtring; a reverse type of 3 is the same as a reverse type of one, and a reverse type of 4 leaves the ſtring in place. Values greater than two are thus pointless, but are allowed for convenience. ❦ Returns: the given ſtart pointer; the ſtring is reversed in place. ❦ Notes: The ſtring muſt be in read-write memory; to reverse a ſtring that was a manifeſt conſtant at compile time, you muſt firſt copy it into a dynamically allocated buffer. ¶ This function is used by lqkwic, which contains some examples.

❦ Errors: An internal error (always fatal) is produced if either start or end is a null pointer, or if end < start (implying a string of negative length), or if the type argument is outside the range from zero to eight inclusive.

---

Utilities/Strings
../liblqutil/strings.c

API int
**LQU_StringContainedIn**(ShortString, LongString)
  CONST char *ShortString;
  CONST char *LongString;

Determines whether the given ShortString is contained anywhere in the LongString, and, if so, returns non-zero. ❦ Returns: 1. 1 if the shorter string is contained in the longer, or if the strings are equal, of if ShortString is of length zero 2. 0 otherwise ❦ Notes: See strstr for a more efficient way to do this. Some Unix systems do not have strstr, though.

---

Utilities/Strings
../liblqutil/cknatstr.c

API int
**LQU_cknatstr**(str)
  CONST char *str;

Checks whether the given string argument represents a natural number; that is, an optional plus or minus sign followed by one or more decimal digits. Leading whitespace, as reported by the isspace macro, is ignored, but no trailing whitespace is allowed. ❦ Returns: Zero if the match fails, and one if it succeeds. ❦ Bugs: 1. Should return a pointer to the first implausible character. 2. Should probably allow trailing whitespace. 3. Does not check its argument for a NULL pointer.

---

Utilities/Strings
../liblqutil/cstring.c

API char *
**LQU_cstring**(theString)
  CONST char *theString

Converts any C escape sequences in the given string, and returns the result in a freshly malloc'd copy. The escape sequences currently recognised are \a (audible alert), \e (escape), \n (newline), \t (tab), \b (backspace), \r (return), \f (form feed), \\ (backslash), \' (single quote) and \" (double quote). The vertical tab (\v) is converted into a newline. The octal \ddd notation is understood; there can be up to three octal digits after the backslash. If you need to follow an octal escape with an ASCII digit, you should use all three digits, with leading zeros if necessary. The ANSI C \xDD hexadecimal notation is not supported. ❦ Returns: A pointer to a freshly allocated buffer; it is the caller's responsibility to free this. If a null pointer was passed as an argument, however, a null pointer is returned. ❦ Errors: Warns if an unrecognised escape sequence or trigraph was found ❦ Bugs: Has support neither for hexadecimal escapes (\xDD) nor for trigraphs (perhaps this is a feature). There is no way to include ASCII NUL (\000) into a string, as this terminates it.

API char *
**LQU_joinstr2**(s1, s2)
  CONST char *s1, *s2;

Utilities/Strings
../liblqutil/joinstr2.c

Returns a string consisting of the concatenation of the two given strings. The result is freshly malloc'd, and it is the caller's responsibility to free this storage. Null strings are treated as if they were empty strings. ❦ Returns: The concatenation of the three given arguments. ❦ Errors: Fatal error if memory is exhausted. ❦ Bugs: The name is a little odd.

❦ Example:

```
char *theFile = LQU_JOINSTR2(directoryName, "/fileName");
```

❦ See Also: LQU_JOINSTR3‡ (below).

---

API char *
**LQU_joinstr3**(s1, s2, s3)
  CONST char *s1, *s2, *s3;

Utilities/Strings
../liblqutil/joinstr.c

Returns a string consisting of the concatenation of the three given strings. The result is freshly malloc'd, and it is the caller's responsibility to free this storage. Null strings are treated as if they were empty strings. ❦ Returns: The concatenation of the three given arguments. ❦ Errors: Fatal error if memory is exhausted. ❦ Bugs: The name is a little odd.

❦ Example:

```
char *theFile = LQU_JOINSTR3(directoryName, "/", fileName);
```