# LifeLines Developer Documentation

---

## LifeLines Version 3.1.1

Perry Rapp

| COLLABORATORS | | | |
|---|---|---|---|
| | *TITLE* :<br><br>LifeLines Developer Documentation | | |
| *ACTION* | *NAME* | *DATE* | *SIGNATURE* |
| WRITTEN BY | Perry Rapp | February 22, 2024 | |

| REVISION HISTORY | | | |
|---|---|---|---|
| NUMBER | DATE | DESCRIPTION | NAME |
| | | | |

# Contents

# Chapter 1

# Introduction to Lifelines Developers Manual

LifeLines source code is divided into several functional subdirectories, which will be discussed individually below. They are chained together by an autotools build system, which creates executables in both the liflines and tools subdirectories.

# Chapter 2

# btree module

The btree subdirectory contains the implementation for a btree database, using fixed length 8 letter keys (RKEY).

**nodes** Each node in the btree is a separate file on disk (named, eg, "aa"), and the first 4096 (BUFLEN macro) bytes are the node header.

**index nodes** These are the interior index nodes of the btree; they contain pointers to subordinate index or block nodes. The program performs binary searches through index nodes to find a particular key.

**block nodes** These contain the actual data (keys and their associated records).

**keyfile** One special file on the disk, the keyfile, contains some meta information and a pointer to the root of the btree (the master key). When the root changes (splits), the master key in the keyfile is updated accordingly.

**traverse** There is a traversal function implemented at the btree level, which uses a callback.

**bterrno** There is a global integer error variable, bterrno, which is set by this module upon most failure conditions.

**FUTURE DIRECTIONS** bterrno must be removed for multi-threading. Traversal is more elegantly done via iterator style repeated calls in, instead of callback.

# Chapter 3

# stdlib module

The stdlib directory contains various utility functions not specifically related to LifeLines, GEDCOM, or even genealogy.

## 3.1 String Functions

There has built up, over time, quite an assortment of string functions, split currently between mystring.c and stdstrng.c (and a few macros in standard.h).

### 3.1.1 String copy and concatenation

char ***llstrncpy**(char *dest, , size_t n, , int utf8, , const char * fmt, , va_list args);

char ***llstrncat**(char *dest, , size_t n, , int utf8, , const char * fmt, , va_list args);

These are simple wrappers around the C RTL (run time library) functions. The ANSI versions do not zero-terminate on overflow, which is greatly inconvenient, os the wrapper versions do so. Also, the wrapper versions are UTF-8 aware (they backtrack on overflow, to avoid leaving part of a UTF-8 multibyte sequence at the end).

### 3.1.2 String append (llstrapp)

char ***llstrapps**(char *dest, , size_t limit, , int utf8, , const char * src);

char ***llstrappc**(char *dest, , size_t limit, , char ch);

char ***llstrappc**(char *dest, , int limit, , int utf8, , const char * fmt);

char ***llstrappv**(char *dest, , int limit, , int utf8, , const char * fmt, , va_list args);

This family of functions is one (thin) layer higher than llstrncpy, providing an interface wherein the caller specified the buffer's start and entire size. That is,

```
llstrncat(buffer, " more stuff", sizeof(buffer)-strlen(buffer));
```

may be replaced by

```
llstrapp(buffer, sizeof(buffer), " more stuff");
```

There are also varargs versions, so that

```
snprintf(buffer+strlen(buffer), sizeof(buffer)-strlen(buffer), ...
```

may be replaced by

```
llstrappf(buffer, sizeof(buffer), ...
```

### 3.1.3 String append (appendstr)

This is a family of functions similar in purpose to the strapp family, but which uses an additional level of indirection, advancing pointers and decrementing counts.

* NOTE: FUTURE DIRECTIONS I put these in, and I would like to take them out, as I find them less intuitive than the strapp family, and more bug-prone. They are slightly faster, but I don't think it is worth it. -Perry.

### 3.1.4 char functions

There are character classification functions, which have handling particular to Latin-1 and to Finnish (if the Finnish compilation option was set).

* NOTE: FUTURE DIRECTIONS It would be very nice to see wchar-based functions, which handle unicode, replace these, and then we might be able to jettison the Latin-1 and Finnish specific character code.

### 3.1.5 string allocation functions

TODO: (strsave, strfree, strupdate, strconcat, free_array_strings)

### 3.1.6 string conversion functions

TODO: (isnumeric, lower, upper, capitalize, titlecase)

### 3.1.7 string equality functions

TODO: (eqstr, eqstr_ex, nestr, cmpstr)

### 3.1.8 string comparison functions

TODO: (cmpstrloc)

### 3.1.9 string whitespace functions

TODO: (trim, striptrail, striplead, allwhite, chomp)

### 3.1.10 string UTF-8 functions

These are the low-level functions used to do UTF-8 mechanics. These should only be called when in a database with internal codeset of UTF-8.

### 3.1.11 printpic functions

These are simple printf style functions, except they only handle string format, and they do handle reordering the inputs. These are used for strings that are internationalized, so that words or numbers (passed in string format) may be reordered in other languages. Instead of %s escapes, these handle %1, %2, and %3 escapes.

## 3.2 List Module

list.c and list.h implement a simple, doubly-linked list type, which takes void pointers (VPTR) as elements. The list manages its own nodes and memory (struct tag_list and struct tag_lnode), but the for the elements, it only frees them if the caller so instructs it (using list type LISTDOFREE), and of course this only works if they are stdalloc/stdfree heap blocks.

## 3.3 Table Module

table.c and table.h implement a fixed size hash tree (with linear buckets). As of 2005-01, Perry has been changing the implementation of the table type, so it is currently in flux.

## 3.4 Balanced Binary Tree (rbtree) Module

rbtree.c and rbtree.h implement a generic red/black balanced binary tree. These are not currently used by lifelines, but are planned as a replacement for the current fixed-size hash table in table.c.

# Chapter 4

# gedlib module

This directory is a collection of routines for GEDCOM and for its use in a LifeLines btree database.

## 4.1 names

This module implements indexing names. TODO: Explain soundex indexing.

## 4.2 refns

This module implements indexing references (REFNs). TOD: Explain two character index.

## 4.3 xreffile

This module stores lists of deleted record numbers for each type. When a record is deleted, its number is added to the appropriate deleted list in xreffile. When a record is added, first the appropriate deleted list in xreffile is checked for a free record number.

## 4.4 messages

Traditionally all translatable strings have been stored in this file. This is not necessary with the current gettext scheme, but it would perhaps be helpful if a resource based scheme were adapted in the future.

* FUTURE DIRECTIONS When/If GUI versions are incorporated into the same codebase, how to handle translate strings shared and not shared between versions needs to be worked out.

## 4.5 translation tables (charmaps.c and translat.c)

The implementation of codeset translation is stored here (not to be confused with language translation for the user interface, called localization, and not associated with these files). Both custom translation tables and delegation to the iconv codeset conversion library are done here.

## 4.6 indiseq

The indiseq type is implemented here, a list of records (which no longer need all be persons).

## 4.7 brwslist

Named browse lists are implemented here (temporary record lists named by user during this session).

# Chapter 5

# interp module

The LifeLines reporting language parser and interpreter are stored here. A custom lexical analyzer is in lex.c, and a yacc parser generator is in yacc.y.

The main interpreter is called with a list of files to parse, and some options. In actuality, I don't think more than one file is ever passed to the main entry point. If no file is passed, the routine will prompt (and here is where the user may choose a report from a list). But a report may be passed in, if one was specified with commandline argument to llines or llexec.

The report file is parsed, and as it is parsed, any included reports are added to the list to be parsed (unless already on the list, so circular references are not a problem).

require statements are handled at parse time. The handler puts the requested version into the file property table (stored inside the pointer in the filetab entry for the file; filetab entries are indexed by full path of report). Later, just after parse completes for that file (in the main parsing loop in the main interpreter function), require conditions are tested in check_rpt_requires(...).

## 5.1 pvalues

All variable values in report language interpretation are stored in a union type called pvalue.

## 5.2 symtab

Symbol tables are a thin wrapper around the table type provided by stdlib, specialized to hold pvalues.

## 5.3 date

A fairly complete GEDCOM date parser is also located here. It actually includes both a date parser, and a date formatter (which generates the thousands of possible LifeLines date formats).

* FUTURE DIRECTIONS If a date type were added to the report language, it would be possible to distinguish fully-parsed dates in the report language (so invalid or illegal dates could be flagged and handled separately in a report). The date module already implements a date type internally, and it is exposed to the rest of the program (gdate and gdate_val, which correspond to GEDCOM date types), but not to the report language.

# Chapter 6

# liflines module

TODO:

# Chapter 7

# autotools build system

todo

# Chapter 8

# Building LifeLines

This chapter gives an overview of one way you can build LifeLines. It is not intended to be a comprehensive list of all techniques, but rather enough to get you started. This section does not assume you are downloading the source tarball and building it, Those instructions are in the file INSTALL. We are assuming you are checking out the sources from CVS.

## 8.1 Cloning the source tree

Anyone can clone the LifeLines source tree, using the following commands:

```
git clone https://github.com/lifelines/lifelines.git
```

Once you have cloned the sources, git hides information in the .git subdirectories so all information about the repository is retained. After the initial clone, if you want to update your sources, you can just type:

```
git pull
```

If you want to contribute (check-in) code to the main repository, please contact Marc Nozell (see github for contact information) to gain project access. Once you do this, you can submit your changes using the following commands:

```
git add <filename>    # to add files
git remove <filename> # to remove files
git commit                  # to commit changes
git push                    # to push files to remote repository
```

## 8.2 automake and autoconf

Many of the files you're used to editing by hand are automatically generated by automake and/or autoconf. These include any file named Makefile, Makefile.in, config.h, config.h.in, or configure.

The proper files to modify by hand are configure.ac (if there's something new you need to determine about the host system at configuration time) and Makefile.am (if source files are added or removed, targets added, or dependencies changed).

As long as you have autoconf and automake installed on your system, the Makefiles generated will be able to regenerate any file dependent on a Makefile.am or configure.ac. To regenerate the build system explicitly run the script autogen.sh:

```
sh build/autogen.sh
```

autogen *must* be run after freshly checking a copy of the project out of git -- the files generated automatically are no longer included in the git repository.

autogen.sh does the following: * Calls aclocal to generate aclocal.m4 from acinclude.m4, and populate build/autotools. * Calls autoheader to generate acconfig.h. * Calls automake to generate Makefile.in files from Makefile.am * Calls autoconf to generate configure from configure.ac

## 8.3 configure

From a source distribution package, or a regenerated development environment, the 'configure' script will generate config.h and generate Makefiles from every Makefile.in. At this point your source tree is properly configured for your machine and can be built.

## 8.4 Building the code on Unix/Linux

There are lots of dependencies required to build LifeLines. These include: * C compiler (gcc, clang, etc) * GNU make * GNU bison * GNU flex * GNU autoconf and automake One way to build the code is to make a subdirectory, lets say called bld in your lifelines directory, (where the toplevel Makefile.am is located) and then build all the code there. This keeps the objects and executables out of the source directories. This is the process shown here, and the process used by the build/build_dist.sh script.

```
sh build/autogen.sh
mkdir bld
cd bld
../configure
make
```

This should build LifeLines and leave the results in subdirectories of the the directory bld.

## 8.5 Generating the source tarball

If you have build the code as described above, you can generate the source tarball as follows;

```
cd bld
make dist
```

While this is a source tarball it does contain a number of generated files that make it easier to generate LifeLines from the source tarball and/or package for third-party distribution. This includes the configure script, the makefiles, and the HTML/PDF documentation.

## 8.6 Generating the rpm package

The specification file to build a rpm for redhat linux is included in the git repository. These notes show how you can use this to build the source and binary rpm for redhat linux.

These instructions use techniques described by Mike Harris in a note entitled "Building RPM packages as a non-root user." These were found at http://www.rpm.org/hintskinks/buildtree. At that url was also a tarball that included the files README( the note), .rpmrc and .rpmmacros. The later two files are installed in your home directory. These do alter the default behavior of rpm for you and are not required to build the rpm, however, these instructions will fail.

Make sure there is a line of the form

```
%packager     Joe Blow  <joe@blow.com>
```

In your ~/.rpmmacros file. It is used to put the name and email address of the individual generating the rpm package into the file. Be sure to use your name and email address. If there is a "Packager:" entry in the lifelines.spec file, make sure it is correct, as it overrides the value in your .rpmmacros file.

From the lifelines directory (where the toplevel Makefile.am and the bld directory are, execute the following commands (with appropriate version numbers of course)

```
mkdir ~/rpmbuild
mkdir ~/rpmbuild/SRPMS
mkdir ~/rpmbuild/RPMS
mkdir ~/rpmbuild/BUILD
mkdir ~/rpmbuild/tmp
mkdir ~/rpmbuild/lifelines-3.0.22
cp bld/lifelines-3.0.22.tar.gz ~/rpmbuild/lifelines-3.0.22.
cp build/rpm/lifelines.spec ~/rpmbuild/lifelines-3.0.22
cd ~/rpmbuild/lifeines-3.0.22
rpmbuild -ba lifelines.spec
```

The mkdir commands only need to be executed if needed. If everything goes ok, this will generate a source and binary rpm.

## 8.7 Making a release

To release a new version, run the build/setversions.sh script to set the version in the many necessary files. Add an entry mentioning the new version in the

```
  ChangeLog
```

Tag the git source via (for example, for version X.Y.Z)

```
  git tag vX_Y_Z
```

Finally, Send an announcement to the LINES-L mailing list

## 8.8 Putting a release on github

This process is still being developed.