libriff: a library that handles RIFF files

Petros K. Tsantoulis

20/08/2001

<ptsant@otenet.gr>
http://avilibs.sourceforge.net

Contents

1 What's this?

This library tries to facilitate the handling of RIFF files, especially AVI (Audio/Video Interleave) files. It is part, indeed the core, of a larger project that will try to offer an easy to use solution for reading and writing of video and audio streams. I am releasing this library to the general public under the popular GNU GPL v.2 license in the hope that it will be useful to free software developers. I reserve the right to change the licensing policy or restrict it to GNU GPL v.2 if later versions of the GNU GPL are not satisfactory.

This is a document that tries to give some information to future developers and users of *libriff*. It is not particularly thorough or complete at the moment. This library and especially its documentation are a work in progress. Feel free to bother me with e-mail at `<ptsant@otenet.gr>` for anything related to *libriff* (spam does **not** count as related.)

2 Installation

Installation should be relatively painless. This library depends only on the standard C library. The GNU C library version 2.2.3 has been used and should be considered the library of reference.

After uncompressing the sources in an appropriate directory (I must assume you have already done that) all you have to do is issue "make ; make install" and the library will be installed under /usr/local/lib and /usr/local/include/libriff. Check the Makefile if you prefer some other destination directory. If you wish to rebuild the current manual from the .tex source you are going to need LaTeX but this is not strictly necessary.

3 General Design

3.1 Object oriented

This library has been written in an implicitly object oriented manner. This means that although C is not an "object oriented" language you are expected to treat all internal data structures as "private" and manipulate them solely with the aid of the accompanying funcitons, unless of course you *know* what you are doing (in which case, C gives you enough rope to hang yourselves, if you so desire.)

3.2 Extending the library

If further basic functionality is needed it can be implemented either on top of the existing functions or on top of the existing data structures (**struct** `riff_handle` .) If data structures are externally modified you should take great care at all times to preserve data integrity. Please note that you may *read* values off the riff_handle structure but this structure may change in future versions while the function interface (if it exists) for the variable of interest should remain consistent. All functions do some sanity tests to ensure that the library maintains a correct and consistent internal environment.

3.3 The C language

The language C was chosen instead of C++ because it generally compiles better, is compatible with C++ and results in smaller and faster executables. It can be argued that C++ has several advantages, but this is a matter of personal opinion and not subject of debate.

Currently the library is faithful to the C89 coding standard, possibly with some GNU C compiler extensions, but some C99 features may be implemented in the feature. GCC v3.0 provides some C99 support and should be considered the reference C compiler. Older compilers should normally work without problems.

4 Bugs, Features and Design Limitations

4.1 Bugs

No known bugs at the moment. Many of them propably exist but I haven't seen them. Do contact me at `<ptsant@otenet.gr>` if you see them.

4.2 Features

Several. I consider the most important to be :

Robustness. This library contains internal checking code. It should handle most errors gracefully and, most importantly, it allows you to easily spot possible bugs.

Portability. This library aims towards maximum portability. It uses standard C library functions and should compile on almost any platform. Effort has been made to make it portable to big-endian machines but

since the author does not use such a machine for developement the code has been untested.

Freedom. As in "Freedom of Speech", this library is released to the general public under conditions that permit and encourage the sharing of source code and ideas. This is a distinct advantage.

4.3 Design Limitations.

There are some design limitations that may be troublesome in some cases. I consider some of them to be (feel free to add your own) :

No internal buffering: This library does not try to buffer data to memory when reading or writing a file. This task is left to the C library functions. I consider standard C library functions to be sufficiently fast when coupled with an operating system that uses reasonable caching techniques. I cannot supply you with measurements that prove my decision was correct but can speculate that the extra speed will not be worth the added complexity.

Limited number of open lists: This library will not handle RIFF files that contain more than 1024 (an arbitrary number that can easily be redefined in "defines.h") open, nested lists. Any number of lists may be contained in the file but they may not be nested deeper than 1024 at a time. I consider this to be a trivial limitation. Indeed it is not hard to add a dynamically growing data type such as a linked list that will solve this problem but this will be relatively slower. I really do not expect RIFF files with more than 1024 nested lists to exist.

Designed to handle AVI files: The code was developed with AVI files in mind and may not be appropriate for other types of RIFF files. The author has made every effort to keep all functions generic and relatively extensible but implicit assumptions could possibly limit the usefulness of this library with other types of RIFF files. Please contact me to report such cases.

No read/write mode: The code will only handle reading *or* writing a file but not both at the same time. Editing a RIFF file in place is quite difficult and cumbersome. You may achieve the same functionality by using a read-only master file and a temporary output file. This allows for cleaner code and more elegant design.

No indexing: It is theoretically possible to build a mechanism for efficient traversal of the file by creating an index structure that refers to the position of all (or perhaps the most recently visited) blocks in the file. Its useability depends on the underlying type of RIFF and I think it should be part of specialized libraries.

Single RIFF structure per file: It is theoretically possible to add more RIFF structures after the end of the first, but *libriff* does not support reading or writing this. You can use multiple files for that. After all, *libriff* properly handles 4GB RIFF files, which is the theoretical maximum, and also a very large file size for many filesystems.

5 Usage

The main features of the library will be presented in a brief manner here. The header and object source code should be considered the definite reference. You may be interested in reading the example code supplied with the library sources.

5.1 Basic RIFF structure

The fundamental blocks of RIFF files are *chunks* and *lists*.

```
CHUNK structure
<4 byte chunk type> (can be '00dc', 'strf' etc)
<4 byte chunk data size>
I: size counted from here
<chunk data>

LIST structure
<4 byte word LIST ('L','I','S','T')>
<4 byte list size>
I: size counted from here
<4 byte list type>
<blocks that belong to the list>
```

Please note that all blocks begin on a 16-bit boundary. The recorded list or chunk size only refers to the data that follows and does not take into account the header.

It is also apparent that the RIFF file is a forward singly-linked structure, in the sense that even though it is relatively easy to seek forward, by

jumping to the next block, it is relatively difficult to seek backward. Indeed, an efficient method of moving backwards has to be implemented by the calling program, perhaps in the form of an index structure. Currently this library does not support such an indexing method. Some types of RIFF files, most importantly AVI files, contain an appropriate index structure (the `idx1` chunk) that can be used to seek in the file. I believe that this kind of indexing should be implemented in an outer layer that corresponds to the RIFF file type of interest and not inside the generic core RIFF library.

5.1.1 Important constants

Some constants that are part of the RIFF file specfication are:

```
/* RIFF file signature */
const uint32_t opRIFF=FOURCC('R','I','F','F');
/* LIST signature */
const uint32_t opLIST=FOURCC('L','I','S','T');
/* JUNK chunk signature */
const uint32_t opJUNK=FOURCC('J','U','N','K');
```

These constants are defined inside the file "riffcodes.h". To build your own see the file "fourcc.h".

5.2 Opening files for reading

The library cannot handle a read/write mode. You can only read or write at a file.

5.2.1 RFRopen_riff_file()

```
riff_handle * handle;

handle=RFRopen_riff_file(``filename'');

if (handle==0)
exit(-1); /* Could not open file */
else
...  /* File succesfully opened for reading */
```

A four character code in the form of a `uint32_t` must be supplied to denote the type of RIFF file that is being created. For example, AVI files are of type `const uint32_t opAVI=0x20495641; /* = 'A','V','I',' ' */`

See the ‘‘`fourcc.h`’’ facilities for building `uint32_t` four character codes from four characters.

Notice that the RFR prefix stands for Riff Read, and the RFW prefix stands for Riff Write. An underscore character "_" may be prepended to show that the corresponding function is internal to the library. Such functions should not be accessed by the calling program under normal circumstances.

5.2.2 RFRclose_riff_file()

```
int
RFRclose_riff_file(riff_handle *handle);
```

When you are done reading the file you can close it (and free the data structures associated with the handle) by calling RFRclose_riff_file(). The handle will no longer be accessible after that.

5.3 Reading blocks

To read a block you need very few functions. The most important are:

```
int RFRget_block_info(riff_handle *handle,
      uint32_t *type,
      uint32_t *size,
      uint32_t *list_type);
int RFRget_chunk(riff_handle *handle,
      unsigned char *buffer);
int RFRskip_chunk(riff_handle *handle);
```

5.3.1 RFRget_block_info()

The most important function is perhaps RFRget_block_info(). You need to call RFRget_block_info() before every action that you plan to take, unless, of course if you *know* that this action is applicable under the current circumstances.

RFRget_block_info() returns the size of the current block and also copies relevant information to the provided variables. The variable `type` contains the type of the next block which is either `LIST` or the chunk type. The variable `size` contains the size of the next block. The variable `list_type` refers to the list type of the current block, *if* it is a list. If the current block is not a list then `list_type` refers to the type of the list that contains it.

The function will not fail if pointers to type, size and list_type are not provided. It will return the current block size only.

5.3.2 RFRget_chunk()

This function copies the data from the current chunk to the provided buffer. This function will report an error if you do not provide an initialized buffer.

Please remember that *no buffer overflow checking is done* and it is therefore possible that your program will perform a segment violation if the buffer is not at least of size equal to the current chunk's size. To avoid such errors you should either use a sufficiently large buffer or ensure at all times that the current buffer is larger than the size reported by RFRget_block_info for the current chunk.

The return value is the number of bytes copied.

5.3.3 RFRskip_chunk()

This function proceeds to the next chunk without handling the current one. It is assumed that the current block is always a chunk. In other words, RFRskip_chunk() will never skip a list. You may use this function to skip chunks that you do not wish to process.

Return values are 0 for success, 1 if the current block is a list and < 0 on various failures.

5.4 Reading lists

Several functions are useful when reading lists:

```
int RFRenter_list(riff_handle *handle);
int RFRskip_list(riff_handle *handle);
int RFRgoto_list_end(riff_handle *handle);
int RFRgoto_list_start(riff_handle *handle);
```

5.4.1 RFRenter_list()

This function allows you to proceed inside the list. It is applicable only if the current block is a list. The alternative function RFRskip_list(), that is also applicable if the current block is a list, will skip the list header and its contents.

The library does keep track of nested lists (lists inside other lists), up to a maximum depth of 1023 lists. This function does some internal book-keeping to ensure that the library is always accurately aware of the list status.

The return value is 0 on succes, and, as usual < 0 on various failures.

5.4.2 RFRskip_list()

This function is applicable if the current block is a list. Contrary to the RFRenter_list() function it will skip the list header and the list contents.

The return value is 0 on succes, and, as usual < 0 on various failures.

5.4.3 RFRgoto_list_end() and RFRgoto_list_start()

These functions jump out of the list and move either to its end or to its start. The effect is actually that of *exiting* the list without producing a special list end block. The function RFRgoto_list_end() should be treated as a way of skipping a list when inside it, its effect is the same as RFRskip_list() when at the start of the list. Note that these functions are applicable only if the current block is a list.

The return value is 0 on succes, and, as usual < 0 on various failures.

5.4.4 List end special blocks

In order to keep your program aware of the current list status the library will inform you by means of special "phoney" block information whenever a list boundary is crossed. These blocks do *not* exist in the actual file, nor are they part of the RIFF specification. They are produced by *libriff*.

The form of a list_end block is:

```
/* This is a list */
block_type     = opLIST;

/* This is the true type of the list that just ended! */
block_size     = list_type;

/* This list appears to be of type ``list'' */
block_subtype  = opLIST;

/* Check for a list_end block */
RFRget_block_info(handle, type, size, list_type);

if (type==opLIST) {
     if (list_type==opLIST)
        /* This is a special list end block. */
        /* The list of type ``size'' just ended. */
     else
        /* This is a normal list */
```

```
} else
     /* This is a chunk */
```

5.5 Seeking

Many functions are devoted to seeking. Most propably you won't need every one of them. As a matter of fact, I strongly suggest that you avoid them, with the exception of RFRfind_chunk() and RFReof(). Improper use of seeking can disrupt the internal data structures and make the library fail. Great care must be taken not to exit or enter any lists without calling the appropriate list functions.

```
int
RFRseek_forward_to_chunk_type(riff_handle *input,uint32_t type);
int
RFRseek_forward_to_list_type(riff_handle *input,uint32_t type);
int
RFRfind_chunk(riff_handle *input, uint32_t type);

int
RFRseek_to_position(riff_handle *input);
int
RFRsave_position(riff_handle *input);
int
RFRrestore_position(riff_handle *input);

long
RFRget_position(riff_handle *input);
int
RFRrewind(riff_handle *handle);

int
RFReof(riff_handle *input);
```

5.5.1 RFRseek_forward_to_chunk_type()

This function and its brother RFRseek_forward_to_list_type() will do exactly what their names imply. They will proceed in a forward direction until they find a block of the appropriate type and they will stop there, returning a value of 1. If the block cannot be found these functions will reach the end of the file, entering and exiting lists in the process.

The most appropriate way of recovering from these functions is RFRrewind(). Simply seeking to original point in the file will *NOT* do because the list keeping has changed and the library will fail. This is a direct result of the inability to move backwards in RIFF files.

5.5.2 RFRfind_chunk()

This is the best function to use for searching. Contrary to the previous functions, this function will *NOT* exit the current list. If it does not find a chunk of the required type it will stop at list end, file end or at the start of a new list inside the current one, whichever occurs first. Upon completion a return value of 1 means that the chunk was found, 0 that the search had to stop. As always, < 0 represents various failures.

5.5.3 RFRseek_to_position()

This function will move the file pointer to the specified position. It will do this in a safe manner. You should not use fseek(). Note that RFRseek_to_position() may need to RFRrewind() the file and can be *very* slow in degenerate cases. It is my understanding that in most cases it will work sufficiently fast. *Please note that if the specified position is not the start of a block, bad things will happen. It is a critical assumption that the file pointer always lies at the start of a block.* As usual, 0 means success and < 0 represents various failures.

5.5.4 RFRsave_position() and RFRrestore_position()

The function RFRsave_position() saves the current file position in an internal holder variable and RFRrestore_position() will go back to that position.

5.5.5 RFRget_position()

This function acts as a wrapper for ftell(). The return value, of type `long` is the file position.

5.5.6 RFRrewind()

This function will return the file pointer to position 12L, right *after* the RIFF file header, and properly initialize the internal data structures. This function can be used to start scanning the file from the beggining.

5.5.7 RFReof()

This function returns 1 if the end of the file has been reached, 0 otherwise.

5.6 JUNK chunks and alignment

Files of RIFF type are implicitly aligned on a 16-bit boundary at the start of every block. However, it is commonly preferrable to align these files to some other boundary such as 2048 bytes (data CD block size) by placing all or some of the blocks in positions that are multiple of 2048.[1] This is achieved by using chunks of type JUNK as padding. See the function RFWalign_riff() for writing aligned blocks.

The *libriff* library has the ability to transparently *skip* all such chunks or treat them as normal chunks whilst reading. When in "skipping" mode you will never encounter a chunk of type JUNK. All are silently ignored. When in normal mode all JUNK chunks are properly reported and it is up to you to handle them or ignore them.

5.6.1 RFRskip_junk()

You can use the function RFRskip_junk() to specify whether you would like the library to silently ignore all junk chunks.

```
int RFRskip_junk(riff_handle *handle, int skip_junk);

RFRskip_junk(handle, 1);  /* Junk chunks will be ignored */
RFRskip_junk(handle, 0);  /* Junk chunks will be processed */
```

Junk skipping is off by default. Return value is equal to skip_junk if succesful, or < 0 on various failures.

5.7 Opening files for writing

The library cannot handle a read/write mode. You can only read or write at a file.

[1] Note that if you intend to align a file so that it can be read quickly off a CD-ROM you will have to align chunks to 2040 so that the chunk data starts at 2048, after the 8-byte header.

5.7.1 RFWcreate_riff_handle()

```
riff_handle * handle;

handle=RFWcreate_riff_handle(‘‘filename’’, uint32_t file_type);

if (handle==0)
exit(-1); /* Could not open file */
else
... /* File ready for writing. */
}
```

5.7.2 RFWclose_riff()

```
int
RFWclose_riff(riff_handle *handle);
```

Closes the RIFF file and frees memory. Returns -1 on failure, 0 on success.

5.7.3 RFWwrite_chunk()

```
int
RFWwrite_chunk(riff_handle *handle,
               uint32_t type,
               uint32_t length,
               unsigned char *data)
```

This function writes a chunk of type “type” with data of length “length” to the associated file. The return value is equal to the number of bytes that actually get written to the file. Note that this is not necessarily equal to “length”, because it also includes at least 8 bytes for the header and, possibly, a single byte for alignment purposes. A succesful write must return length+8 or length+9. Please remember that the data pointer must have been properly initialized. If the function fails you will get a return value < 0.

5.7.4 RFWrewrite_chunk()

```
int
RFWrewrite_chunk(riff_handle *handle,
      uint32_t type,
      uint32_t length,
      unsigned char *data,
      long file_pos)
```

Rewrites a chunk at the specified position. This function's only difference with write_chunk() is that it does not increase the file and also it does not do any alignment. Returns number of bytes written, ie length plus 8 bytes for the header. This is a potentially dangerous function. It can disrupt the RIFF file if not used with caution.

5.7.5 RFWwrite_raw_chunk()

```
int
RFWwrite_raw_chunk(riff_handle *handle,
                   uint32_t length,
                   unsigned char *data)
```

Returns number of bytes written, normally equal to "length". The chunks are written "as-is", and may not be legal if you do not take care to supply an appropriate header. No alignment is done after the data, but this function will ensure that they start at an aligned boundary.

5.7.6 RFWcreate_list()

```
int
RFWcreate_list(riff_handle *handle, uint32_t type)
```

Creates a list of type "type". This function opens the list and reserves space for its header. The list size is not written until you close the list with RFWclose_list(). The return value is normally 12. Anything else signifies an error.

5.7.7 RFWclose_list()

```
int
RFWclose_list(riff_handle *handle)
```

Closes the last list that was opened and writes its header properly. Note that you can't have more than MAX_LISTS (1024) lists open at once.

5.7.8 RFWalign_riff()

```
int
RFWalign_riff(riff_handle *handle, int boundary)
```

This function writes a JUNK chunk of the appropriate size to align the file at the requested boundary. Note that the requested boundary should be greater than 8 bytes, because that is the size of the CHUNK header. You may not specify an odd boundary, because RIFF files are word aligned.

5.8 Is that all?

Several other functions also exist but are of limited interest. You are urged to read the header files to see if some of the more esoteric or trivial functions not presented here suit your needs.

6 Notes to developers

If you are willing to work further *on* this library, instead of working *with* it there are some minor points you should perhaps consider.

6.1 Coding style

The coding style is as verbose as possible in order to ensure readability and maintenability. I personally prefer the "stroustrup" coding style that emacs offers. You can see for yourselves by examining the source code files.

According to the GNU coding standards all functions in the library are prefixed to ensure that naming conflicts do not arise. Function names are prefixed with **RFW** if they are useful when writing and with **RFR** if they are useful when reading. An underscore character "_" is added to the front to denote functions that are internal to the library and should not be invoked by the calling program.

Comments should clearly describe what the code does and should mention important events such as the opening of a file, allocation/deallocation of memory etc. I consider a well written comment header before the function declaration an element of good coding style. I tend to copy headers between .c and .h files to keep them up to date.

Another point of interest is the examination of return values. I strongly suggest the checking of all return values, especially when calling functions external to this library, e.g. C library functions. As a general rule, every function should verify its input parameters, the status of the internal data structures and the return values of all called functions.

6.1.1 Sanity checks and verbosity

Debugging checks are done with the aid of the "sanity.h" facilities. Two useful macros are provided : `SCHK(condition, action)` and `XCHK(condition, action)`. An SCHK is enabled whenever CHECKS is defined at compiled. An XCHK is enabled whenever EXTENDED_CHECKS is defined at compile time. Generally speaking, SCHK is used for reasonable and often quick checks while XCHK refers to more computationally expensive checks that

are not likely to fail. The possible actions are ACT_FAIL, ACT_WARN and ACT_ERROR. I try to use ACT_FAIL whenever something has gone bad and cannot be corrected or understood. The default action, then is to exit() the program. This is done, for example, if the internal data structures become corrupt, or if the library cannot allocate memory. An error that is handled with ACT_ERROR is usually less serious, possibly the result of bad calling parameters. In that case, the action is to return(-1) from the current function. The ACT_WARN action means that only a message is printed. A message from SCHK() or XCHK() will contain the appropriate filename, line number and function name to guide you.[2]

6.2 Optimization

This library has *not* been specially optimized for speed. The functions that handle I/O cannot influence performance to a significant extent because these functions do not contain nested loops or deep recursion and are mostly bound by the time it takes to make the physical act of writing to the hard drive/cd-rom/memory etc. Even when reading large files (~ 500MB) the CPU time spent inside the library functions is negligible. Most functions are either sufficiently fast or disk-bound.

If you think that extra speed is required you may try to compile without extended or standard sanity checks. See the file ‘‘`defines.h`’’ for this. You may also omit debugging information (`-g` includes debugging information in GCC) and try some compiler optimization options such as `-O3 -march=<your cpu>` for GCC.

I do not intend to optimize this library for speed at the moment, if ever.

[2]This is of course only supported if __LINE__, __FILE__ and __FUNCTION__ are properly defined by your compiler. Most probably they are.