libavi: a library that handles AVI files

Petros K. Tsantoulis

11/01/2002

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

Version **0.69**

Contents

1 Introduction

This library is being released to the general public under the terms of the popular GPL version 2. This is a C library of functions that tries to facilitate the manipulation of AVI files. It requires *libriff* to build succesfully. This library does **not decode frame content**. It never will. This library is a **file manipulation library**. It will happily read or write frames (chunks) according to the AVI structure but it will not encode or decode their content. This is a seperate task that will be implemented either in the calling application or in a third library that I plan to develop in the future.

The overall design of *libavi* is quite simple. There is a mechanism for reading and a seperate mechanism for writing. Other than being aware of the AVI file structure this library also offers a semi-automatic *stream support* mode that greatly simplifies reading and writing by handling streams as seperate entities. If you wish to read/write AVI files with *libavi* you have two choices[1]:

AVI file I/O : You can use *libavi* to read/write the headers and then read/write the frames and de-multiplex them in your application. This mode offers some "AVI-awareness" in the sense that you won't have to parse the header and index structure. However, you get full control over the *movi* list and its contents.

Stream support : You can use *libavi* to seek and read frames from individual streams. In this semi-automatic mode the library handles demultiplexing and seeking (and, to some extent stream synchronization) and all you have to do is handle the frame contents yourself. Each stream is a seperate entity.

I decided to build this manual in the form of a tutorial because learning by example is much easier and *libavi* is actually oriented to a specific purpose (unlike *libriff* that is an all-purpose tool and requires an all-purpose manual). For questions of a more technical nature you should consult the header and source files. They are quite thoroughly documented with verbose comments, including function input, output and side effects.

[1]You may also use *libriff* for maximum control over the file I/O, but this means that your application must be able to parse the AVI structure and generate valid AVI files. This is not recommended.

2 Current Status (version 0.69)

Unfortunately, *libavi* is not yet complete. I'd say that quite a lot remains to be done. This section has been added to warn you about what works and what does *not* work in the current version. I decided to start writing this tutorial before completing *libavi* because the current functionality is adequate for many simple tasks and quite possibly some people would like to start working with it, or *on* it.

The current version (**0.69**) does not implement stream support for writing. Writing is much easier than reading because writing need only produce a valid AVI file, while reading should properly handle all valid AVI files. Writing does work but without any kind of automation, e.g. the index is not built automatically, the streams are not automatically synchronized. You should be able to get valid AVI files without much trouble but writing will definitely be automated in the future with the addition of stream support.

Reading can properly parse the header structure, including exotic chunks such as `strn`. Stream support for reading currently allows only seeking by one frame either forward or backward. It will work with or without an index chunk. The current version is also capable of building an index chunk for a file with a corrupt index or without an index. Please note that building an index for a large AVI file can be time consuming due to heavy disk usage. This is not a bug in *libavi*.

The major problem with the current version and one that must be solved in the near future is the timing of audio streams. I'm unable to properly time audio streams, especially VBR ones. If someone knows how to do this or can point me to an authoritative source please do contact me. Note that the library is able to return audio frames in their proper order and can demultiplex them without problem. The problem is that the `stream_time[]` field in the `avi_handle` structure is currently not reliable for audio frames.

Another minor nuisance is the lack of broader seeking abilities such as seeking to some time point, seeking to the Nth frame etc. This can be circumvented very easily in the calling application by proper usage of the existing functions, but with a small performance hit. These functions will be added in the future.

Note that by design *libavi* will handle multiple streams, up to the maximum theoretical limit of 100 streams. The stream number (00, 01 etc) serves as the only identity that distinguishes its related frames. This means that frames 00dc, 00wb, 00pc will all be reported as belonging to stream 00 regardless of stream type. A proper AVI file should not cause any problems. This is not a bug in *libavi*.

3 Reading AVI files

3.1 Opening the file

The following code will open the file and associate it with the `avi_handle` structure for future reference. It will parse the header and index. Note that, like *libriff*, all function names in *libavi* are prefixed with AVR if they are related to reading and AVW if they are related to writing. A prepended underscore (_) shows that the function is intended for internal use. You may use these so-called internal functions if you need to but they are not really the front end of the library. This small piece of code prints out a lot of interesting information. Try it out and see. It is included under the filename `example1.c`

```
/******* EXAMPLE 1 **********/
/* libavi usage example */

#include <stdio.h>
#include <stdlib.h>
#include "aviread.h"


int
main(int argc, char *argv[])
{
  avi_handle *avifile;
  int i;

  avifile=AVRopen_avi_file(argv[argc-1]);

  /* Print information regarding the main avi header */
  AVRprint_header_info(avifile, stdout);

  /* Print information for all streams */
  for (i=0; i<avifile->total_streams; i++) {
    AVRprint_stream_info(avifile, stdout, i);
  }

  AVRclose_avi_file(avifile);
  return(0);
}
```

3.2 Seeking a little bit

Now we will try to move a little inside the file. But first you need to understand a few things about the way *libavi* sees the file. The library always keeps N+1 file descriptors open[2] for every AVI file, where N is the number of streams. The library actually associates N `riff_handle` structures, one for each stream, with the file and every one of them set to "linear view", i.e. without list information from *libriff*. The remaining `riff_handle` is the main handle that is used to parse the header and index. This handle—accessible through `avi_handle->riff_file`—is not set to linear view and cannot be used for raw file seeking because it will upset list information.

If you decide to do your own I/O all you need to do is use the traditional *libriff* functions such as `RFRget_chunk()` and `RFRskip_chunk()` on the main riff handle (`avi_handle->riff_file`). You may use raw seeking (`fseek()`) if you set it to linear view or you may choose to use the functions from `riffseek.h`.

If on the other hand you use the stream support that the library offers, you only deal with streams numbered from 0 to `avi_handle->total_streams-1` and you don't care about the underlying file.

In the next simple example we will use stream support to read 3 frames from stream 0, skip 2 frames in stream 0, read 2 frames from stream 1 and then go back 1 frame in stream 0. This example expands example 1 and is also available under the filename `example2.c`. All seeking is done via the index chunk provided, so make sure that the AVI file you specify on the command line has an index chunk and at least two streams.

```
/******* EXAMPLE 2 **********/
/* libavi usage example */
/* Call with "example2 <avi filename>" */

#include <stdio.h>
#include <stdlib.h>
#include "aviread.h"


int
main(int argc, char *argv[])
```

[2]This requires a theoretical maximum of 101 simultaneously open file descriptors for every AVI file but this is extremely unlikely to occur in everyday practice. Normally 3 descriptors will be open for every file. The Linux kernel can easily handle 1024 open files.

```
{
  avi_handle *avifile;
  int i;
  int frame_size;
  uint32_t frame_type;

  /* a 128KB buffer */
  char bigbuffer[131000];


  /* Open the file that is passed as argument #1 */
  avifile=AVRopen_avi_file(argv[argc-1]);

  /* Print information regarding the main avi header */
  AVRprint_header_info(avifile, stdout);

  /* Print information for all streams */
  for (i=0; i<avifile->total_streams; i++) {
    AVRprint_stream_info(avifile, stdout, i);
  }

  /* Get three frames... If size 0 then either the
   chunk is indeed 0 bytes long or the file pointer
   is not positioned on a chunk belonging to
   that stream. This may happen under some conditions
   but it will not disrupt the library. The seeking
   functions should be able to re-position the stream
   properly.
  */
  frame_size=AVRget_frame_idx(avifile, 0, bigbuffer, &frame_type);
  AVRnext_frame_idx(avifile, 0);
  printf("First frame from stream 0. Size %d, type [%s]\n",
 frame_size, _RFWfcc2s(frame_type));

  frame_size=AVRget_frame_idx(avifile, 0, bigbuffer, &frame_type);
  AVRnext_frame_idx(avifile, 0);
  printf("Second frame from stream 0. Size %d, type [%s]\n",
 frame_size, _RFWfcc2s(frame_type));

  frame_size=AVRget_frame_idx(avifile, 0, bigbuffer, &frame_type);
  AVRnext_frame_idx(avifile, 0);
```

```
 printf("Third frame from stream 0. Size %d, type [%s]\n",
frame_size, _RFWfcc2s(frame_type));

 /* Skip 2 frames */
 AVRnext_frame_idx(avifile, 0);
 AVRnext_frame_idx(avifile, 0);

 /* Read from the other stream */
 frame_size=AVRget_frame_idx(avifile, 1, bigbuffer, &frame_type);
 AVRnext_frame_idx(avifile, 1);
 printf("Got the first frame from stream 1. Size is %d.\n",
frame_size);

 /* We now choose to ignore frame type information
    (this is OK, if you now what sort of frame you
    are going to get). Simply pass a NULL argument
    and AVRget_frame() will ignore it. */
 frame_size=AVRget_frame_idx(avifile, 1, bigbuffer, NULL);
 AVRnext_frame_idx(avifile, 1);
 printf("Got the second frame stream 1. Size is %d.\n",
frame_size);

 /* Go back 1 frame in stream 0 */
 AVRprevious_frame_idx(avifile, 0);

 /* That's all */

 /* Close the file and free memory */
 AVRclose_avi_file(avifile);
 return(0);
}
```

Note that getting the frame via `AVRget_frame_idx` does not advance the index pointer. If you call it again you will get the same frame, again.

3.3 Seeking without the index chunk

For those that wish to seek without referring to the index chunk the function `AVRnext_frame()` is able to find the next frame that belongs to the requested stream. You may then use the function `AVRget_frame()` to read the frame data for the requested stream. Note that getting the frame *moves* the file

pointer and therefore you may not need to call `AVRnext_frame()` to move to the next frame. A trivial example of this can be found in the test code, inside the test directory. I will not quote this code here because it has not been nicely formatted for postscript output.

Seeking backwards is *not* supported without an index chunk because the RIFF file format is not well suited to moving backwards. I *strongly* suggest that you use the index unless you have a very special reason to avoid it. The functions that do not use the index have not been extensively tested, but they should work without much trouble.

3.4 Building the index chunk

In cases where the index chunk is corrupt or missing you can use the library function `_AVRbuild_idx()` to create an index in memory. This is time consuming because it scans through the whole file. You can then use all the functions that require an index such as `AVRnext_frame_idx()`.

4 Epilogue

Well, that's all for a few months. If this version does not contain major bugs it will not be updated soon. I will be extremely busy in the following months and this project is likely to freeze. I'll propably catch up sometime in the spring. Have fun!