Printed in full color

Quick Start Guide to FFmpeg

Learn to Use the Open Source Multimedia-Processing Tool like a Pro

V. Subhash

Apress®

Quick Start Guide to FFmpeg

Learn to Use the Open Source Multimedia-Processing Tool like a Pro

V. Subhash

Apress®

Quick Start Guide to FFmpeg: Learn to Use the Open Source Multimedia-Processing Tool like a Pro

V. Subhash
Chennai, Tamil Nadu, India

ISBN-13 (pbk): 978-1-4842-8700-2 ISBN-13 (electronic): 978-1-4842-8701-9
https://doi.org/10.1007/978-1-4842-8701-9

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: James Robinson-Prior
Development Editor: James Markham
Coordinating Editor: Jill Balzano

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media LLC, 1 New York Plaza, Suite 4600, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub (https://github.com/Apress). For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Dedicated to the creators and supporters of free and open source software

Table of Contents

About the Author

V. Subhash is an invisible Indian writer, programmer, and illustrator. In 2020, he wrote one of the biggest jokebooks of all time and then ended up with over two dozen mostly nonfiction books including *Linux Command-Line Tips & Tricks*, *CommonMark Ready Reference*, *PC Hardware Explained*, *Cool Electronic Projects*, and *How To Install Solar*. He wrote, illustrated, designed, and produced all of his books using only open source software. Subhash has programmed in more than a dozen languages (as varied as assembly and Java); published software for desktop (*NetCheck*), mobile (*Subhash Browser & RSS Reader*), and the Web (*TweetsToRSS*); and designed several websites. As of 2022, he is working on a portable JavaScript-free CMS using plain-jane PHP and SQLite. Subhash also occasionally writes for the *Open Source For You* magazine and CodeProject.com.

About the Technical Reviewer

Gyan Doshi has been with the FFmpeg project as a developer and maintainer since 2018. During this time, he has focused on FFmpeg filters, formats, and command-line tools. From his experience in video postproduction stages such as editing and motion graphics, Gyan has learned how FFmpeg can be used in multimedia workflows as a valuable addition or as a substitute for expensive tools. Aside from being engaged as a multimedia/FFmpeg consultant, Gyan also troubleshoots FFmpeg issues on online forums such as Stack Exchange and Reddit.

Gyan builds the official Windows binary packages of FFmpeg (`ffmpeg`, `ffprobe`, and `ffplay`) and other tools (`ffescape`, `ffeval`, `graph2dot`, etc.) and offers them for download from his website at `www.gyan.dev`.

Acknowledgments

The author would like to thank:

- The publisher Apress who insisted on not using any third-party video in the screenshots, as the author did in the original self-published book (*FFmpeg Quick Hacks*). Most screenshots in this Apress book were taken from the author's own videos. The rest used videos and images that were in the public domain (Archive.org, Pixabay.com, and Unsplash.com). This led to a rewrite of most of the content, and in the process, several mistakes were eliminated.
- The technical reviewer Gyan Doshi for pointing out several other mistakes and making valuable suggestions.
- Creators and supporters of *free and open source* projects.
- The author's family, friends, enemies and governments without whose help and encouragement this book would have been completed much ahead of its deadline.

Introduction

FFmpeg is a *free and open source program* for editing audio and video files from the command line. You may have already known FFmpeg as a nifty program that can do simple conversions such as:

```
ffmpeg -i some-video.mov same-video.mp4
ffmpeg -i song-video.mp4 song-audio.mp3
```

FFmpeg is much more capable than this, but it is this intuitive interface and support for a wide variety of formats that has won it millions of users.

The FFmpeg project was originally started by a French programmer named Fabrice Bellard in the year 2000. It is now being developed by a large team of open source software developers spread around the world.

This book can serve as an easy FFmpeg tutorial, hack collection, and a ready reference. However, it is not possible for one book to cover everything that FFmpeg can do. FFmpeg has a very huge online documentation with which you may have to craft your commands. While this book may seem more than enough for most users, the documentation will open up vastly more possibilities. DO NOT avoid going through the documentation.

Before you go further into the book, you should be aware that the FFmpeg project creates two types of software:

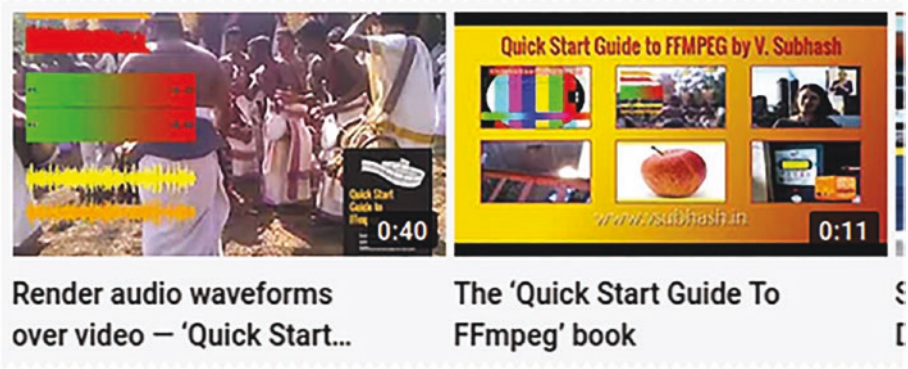
1. `libav` **libraries**: These are FFmpeg programming software or "libraries" that are used by programmers to create audio/video processing software such as media players, browser plug-ins, and audio/video editors. The `libav` libraries have been used to build some parts of popular software such as VLC, xine, Blender, and Kodi.

2. `ffmpeg` **command-line program**: This is the FFmpeg end-user software that most people can use. The `ffmpeg` *command-line program* internally uses the `libav` *libraries*.

In this book, we will ignore the `libav` *libraries* and instead focus on the `ffmpeg` *command-line program*.

Extra Resources for This Book

- All code snippets used in this book are available in a plain-text file, complete with chapter and section titles and comments. It is actually a MarkDown/CommonMark file. You can easily convert it to an HTML, ODT, DOCX, or PDF file. Conversion instructions are in the text file.
- Videos of several code examples used in the book are available in an online video playlist.



Links to these resources can be found at

- www.apress.com/9781484287002 (domain + ISBN)
- www.vsubhash.in/ffmpeg-book.html

CHAPTER 1

Installing FFmpeg

In the Introduction, I mentioned that FFmpeg was an "end-user program." It is actually three command-line end-user programs, or **executables**:

1. `ffprobe`
2. `ffplay`
3. `ffmpeg`

The executables for these programs are available for Linux, Mac, Windows, and other operating systems (OSs). When you go to the FFmpeg website (`www.ffmpeg.org`), you will have two download options:

- Either download **pre-built FFmpeg executables** to your computer
- Or download **FFmpeg source code** to your computer and build your own customized FFmpeg executables

If you are unfamiliar with building executables from source code (as are most people), you should choose the first option.

FFmpeg for Microsoft Windows Users

The download options on the FFmpeg site for *pre-built FFmpeg executables* change frequently, so this book will not be specific with instructions. Just go to this page and navigate to one of the download sites.

```
https://ffmpeg.org/download.html
```


V. Subhash, *Quick Start Guide to FFmpeg*, https://doi.org/10.1007/978-1-4842-8701-9_1

On the selected download site, you may be presented with a dizzying array of downloads. Spend some time reading the information given there, and pick the most appropriate download for you.

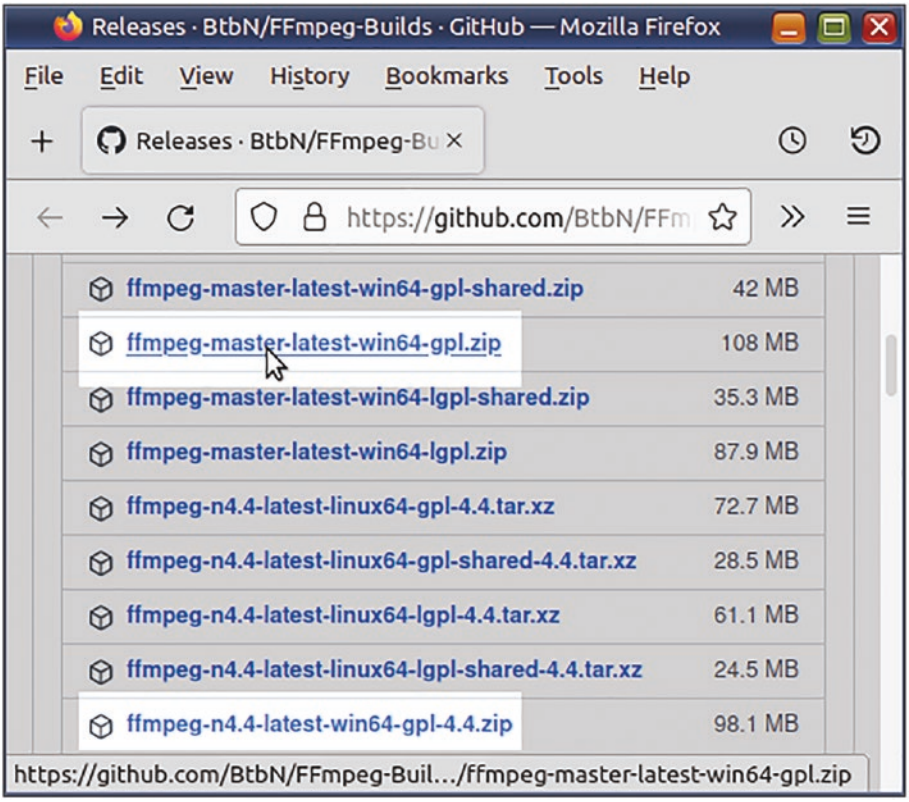


Figure 1-1. *This download page lists several download options for FFmpeg executables. Strangely, for FFmpeg, the latest master download is supposed to be more stable than the numbered release version*

Sometimes, there may be an *essentials* build and a *full* build. The *essentials* build may be enough for most people. If you want to use certain unusual features such as *frei0r* filters, you should choose the latter. As you never know what you might need in the future, I suggest that you choose the *full* build.

Figure 1-2. *There may be more than one "build" option for the downloads*

In the downloaded archives (zip or 7z files), you will find the executables: `ffprobe.exe`, `ffmpeg.exe`, and `ffplay.exe`.

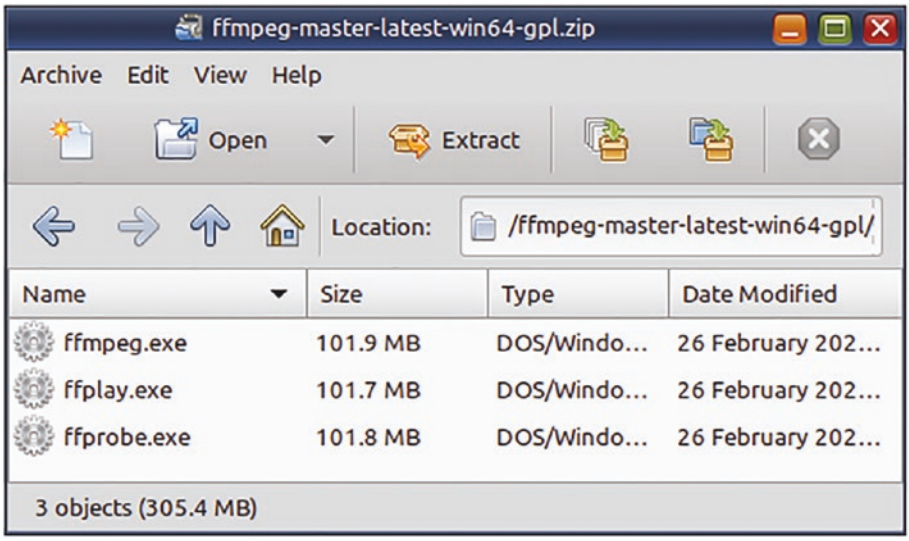


Figure 1-3. *The downloaded archive file contains three EXE files. Copy them to a folder specified in your PATH environment variable*

Copy the EXE files to some folder that is already included in your operating system's PATH environment variable. If you copy them to a new folder, then add the folder's full location to the PATH variable.

If you do not do the above, you will need to type the full path of the executable in your commands in the *Command Prompt* window.

Before modifying the PATH environment variable, take a backup of its value. Open the **Command Prompt** window and type this command.

```
echo %PATH% > PATH-BAK.TXT
```

Let us assume that you have extracted the EXE files to the folder `C:\MyInstalls\ffmpeg\bin`. Launch the **Command Prompt** window with Administrator privileges. Then, permanently suffix this folder's location to the PATH environment variable with this command.

```
SETX /M PATH "%PATH%;C:\MyInstalls\ffmpeg\bin"
```

Then, you should check whether the FFmpeg installation is accessible from the command-line without the full path. (Do this in a **Command Prompt** window with normal-user privileges.)

```
ffmpeg -version
```

If you do not modify the environment variable, then you will have to type the full path whenever you want to use the program.

```
C:\MyInstalls\ffmpeg\bin\ffmpeg -version
```

FFmpeg is case-sensitive so do not type `FFMPEG -VERSION` and hope to get a correct response. FFmpeg may have become platform-independent, but in its heart, it still beats like a Linux program. This means that FFmpeg will not support certain functionalities expected of native Windows/DOS programs. For example, you cannot type command switches (arguments) in uppercase (even if the command name can be typed in uppercase).

```
@ Causes error
FFMPEG -VERSION

@ Causes no error
FFMPEG -version
ffmpeg -version
```

Almost all command-line examples in this book assume a Linux environment. One-line commands will not require any change in Windows.

The Windows/DOS counterpart for the Linux null device (`/dev/null`) is `NUL`. This means that you should replace all instances of `2> /dev/null` in this book with `2> NUL`. This construct is used to prevent the commands from displaying text messages on the screen. `ffmpeg` outputs all its messages to *standard error*, which happens to be the screen. In case it outputs something to *standard output*, which also happens to be screen, and has to be blocked, the Linux remedy is to use `> /dev/null`. To do the same on your Windows computer, you will have to use `> NUL` instead.

In multiline commands, you will find a "\" (backslash) at the end of each line (except the last one), as is the practice in Linux.

```
# For 'nix users
ffmpeg -f lavfi \
       -i "testsrc=size=320x260[out0];
           anoisesrc=amplitude=0.06:color=white[out1]" \
       -t 0:0:30 -pix_fmt yuv420p \
       test.mp4
```

As a Windows user, you should use a caret (^) instead of the backslash (\ ).

```
@ For Windows users
ffmpeg -f lavfi ^
       -i "testsrc=size=320x260[out0]; ^
           anoisesrc=amplitude=0.06:color=white[out1]" ^
       -t 0:0:30 -pix_fmt yuv420p ^
       test.mp4
```

You should avoid writing anything after the backslash or the caret. Invisible trailing space(s) can also make a command to fail. (This happens often with copy-pasted commands.)

In a Linux `bash` terminal, the backslash is not required after a double-quotation mark has been opened, and you can continue on like that for more lines until the quotation is closed. In a Windows `cmd` terminal, all wrapping lines will have to end with a caret.

FFmpeg for Linux Users

If your Linux distribution has not installed FFmpeg by default, then use its default *software manager* or *package manager* to do so. Beware that the FFmpeg installed from software repositories used by Linux distributions are usually out of date.

The download sites linked by FFmpeg.org provide the latest builds with maximum support for external libraries. However, some Linux users like to build their executables from source. If you have a fast machine or a few hours to spare, start with the instructions on the *FFmpeg Wiki* site. Check their *source code compilation steps* specific to the Linux distribution that you use.

```
https://trac.ffmpeg.org/wiki/CompilationGuide
```

You can customize your FFmpeg build by enabling/disabling several build options. Instead of just blindly following the wiki, spend some time studying the `configure` script or its help output.

```
configure --help
```

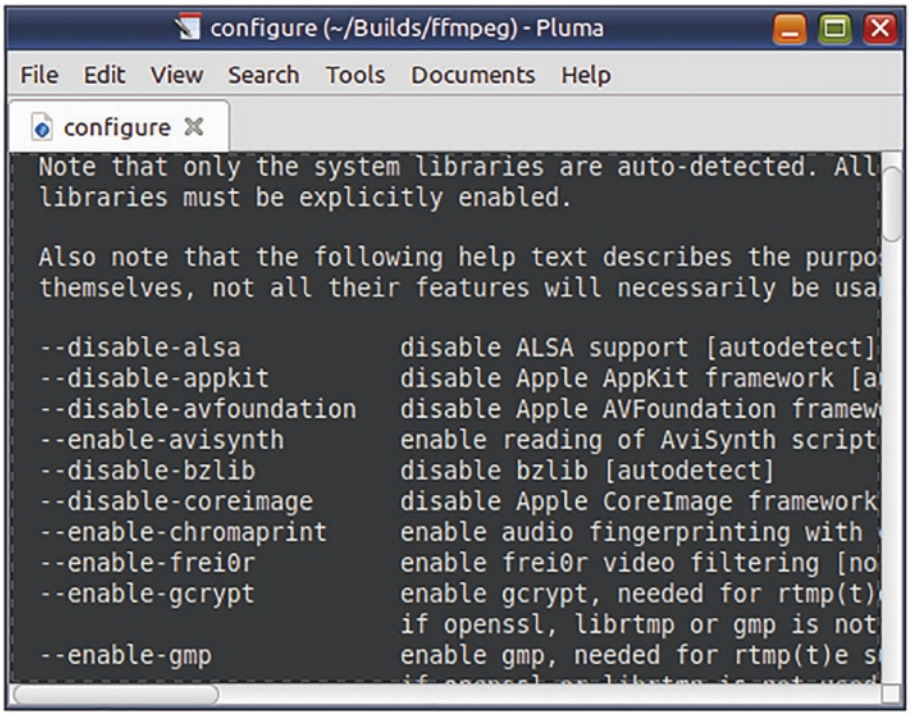


Figure 1-4. *The* `configure` *script, by default, will try to autodetect external libraries. You may have to manually enable those that are not autodetected*

In your Linux package manager app, try to search and install (*dev*-suffixed) developmental packages with similar names as the external libraries. You may not be able to install developmental packages for all of the libraries. But, for whatever libraries that you can install or have them already installed, add relevant `-enable` options to the `configure` compilation step. Here are a few:

```
...
--enable-chromaprint --enable-frei0r \
--enable-libbluray --enable-libbs2b --enable-libcdio \
--enable-libflite --enable-libfontconfig \
```

```
--enable-libfreetype --enable-libfribidi \
--enable-libmp3lame --enable-libsmbclient \
--enable-libv4l2 --enable-libvidstab \
...
```

Run the FFmpeg build statement with these changes, and eventually all three binary executable files will be created in your `$HOME/bin` directory. Then, secure the copy of the documentation from the `ffmpeg_build` directory so that you can read it whenever it is required.

☞ When I built FFmpeg version 5.1, I encountered some errors with the official wiki guide. The guide uses one long stringified command to install the FFmpeg binary executable files. This command is a combination of several commands that downloads the source and then configures, compiles, builds, and installs the executable files. If the configuration and compilation commands encounter any errors and you fix it, the command will restart the whole drama beginning with downloading the source. You do not have to endure that. Just continue with the `configure` step.

If you have an old OS where the latest FFmpeg executable does not run or cannot be compiled, go to `https://johnvansickle.com/ffmpeg/` and download pre-built statically linked executables (not including `ffplay`). On my old Ubuntu 10 *Fiendish Frankenstein* installation, I could not run the latest FFmpeg pre-built executable nor build the source, but these statically linked executables worked. (Even the C library is statically linked.) That is how I was able to finish the 2020 version of this book in the old OS.

FFmpeg for Apple Mac Users

With Apple moving from Intel x86 to ARM architecture, any specific instructions will be outdated when you read it. It is best that you consult the FFmpeg Wiki for the specific kind of Apple hardware that you are using.

```
https://trac.ffmpeg.org/wiki/CompilationGuide/macOS
```

Summary

Although originally designed as a Linux program, FFmpeg is also available for Windows and Mac operating systems. In this chapter, you learned how to obtain pre-built FFmpeg executables specific to your OS from the official FFmpeg site. You also learned how to build your own customized FFmpeg executables from source.

In the next chapter, you will learn how to start using the executables.

CHAPTER 2

Starting with FFmpeg

The FFmpeg project provides several end-user programs. This book will focus on three command-line programs – `ffprobe`, `ffplay`, and `ffmpeg`. You will be using `ffmpeg` most of the time, but `ffprobe` and `ffplay` can help you as well. In this chapter, you will gain an introduction to all three.

All three have an annoying "feature" – they display a build-information banner that is as big as the state of Texas. If you create the following aliases in your `$HOME/.bashrc` file, then you do not have to suffer the annoyance.

```
alias ffmpeg='ffmpeg -hide_banner '
alias ffplay='ffplay -hide_banner -autoexit '
alias ffprobe='ffprobe -hide_banner  '
```

☞ The `-autoexit` option for the `ffplay` command ensures that it makes a clean exit after playing a file instead of sticking around like it has crashed.

Some command examples in this book will have the suffixes `2> /dev/null` or `> /dev/null`. Such recourses were necessary to prevent information clutter.

V. Subhash, *Quick Start Guide to FFmpeg*, https://doi.org/10.1007/978-1-4842-8701-9_2

ffprobe

If you want to find out useful information about an audio or video file, you need to use `ffmpeg` with the `-i` option. With `ffprobe`, you do not need the option.

```
ffmpeg -i tada.wav
ffprobe tada.wav
```

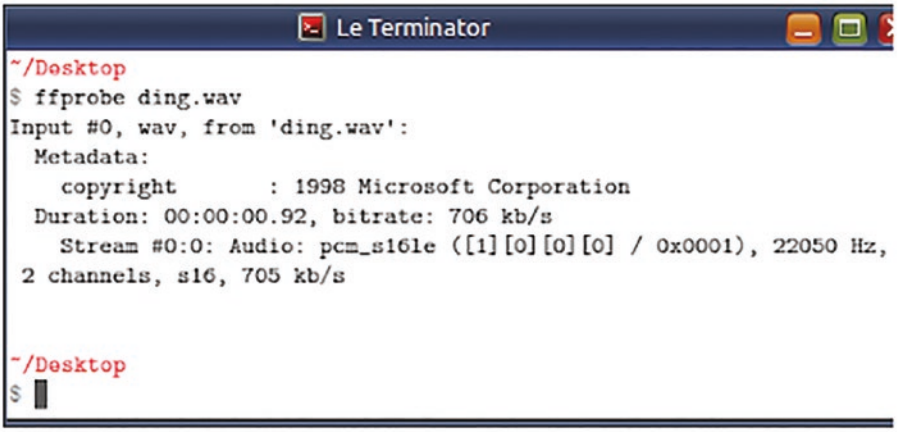


Figure 2-1. *ffprobe can be used to display information about what is contained in a multimedia file*

`ffprobe` can reveal much more information than this if you use the `-show_streams` option. You can filter the output of this command for use in your shell scripts. In a later chapter, you will find a sample output of this command.

```
ffprobe -show_streams somefile.mp4
```

ffplay

If you want to play a video file directly from the command line, just type `ffplay` and the file name. `ffplay` is a tiny media player. It does not have a context menu system or other interface. It responds to some keys and mouse clicks but does nothing more.

```
ffplay solar.mp4
```

Figure 2-2. *`ffplay` can be used to play audio and video files*

To play an audio file without the (windowed) interface, say, as an audio notification in a shell script, you can use `ffplay` like this:

```
ffplay -autoexit -nodisp ding.wav
```

ffmpeg

The executables `ffprobe`, `ffplay`, and `ffmpeg` have several common command-line options (arguments, switches, or parameters). You can list most of them with the `-h` option.

```
ffmpeg -h
ffmpeg -h long
ffmpeg -h full > ffmpeg-help-full.txt
```

If you want to review some of the features supported by your installation of FFmpeg, try these:

```
ffmpeg -formats
ffmpeg -encoders
ffmpeg -decoders
ffmpeg -codecs
ffmpeg -filters
```

The output of these commands will give you a good overview of what FFmpeg can do. Sample output of these commands is available as annexures in this book.

You can dig out more specific help information with commands such as these:

```
ffmpeg -h demuxer=mp3
ffmpeg -h encoder=libmp3lame
ffmpeg -h filter=drawtext
```

Other FFmpeg End-User Programs

The FFmpeg project provides a few other command-line tools in addition to the three introduced in this chapter. Their purpose and usage are beyond the scope of this book. If you wish to do your own R&D, then you can find their files at `www.gyan.dev/ffmpeg/builds/#tools`.

Summary

In this chapter, you gained an introduction to the three FFmpeg executables. Before venturing into what FFmpeg can do for you, you need to learn a few things about multimedia formats and codecs. The next chapter will help you with that.

CHAPTER 3

Formats and Codecs

An MP3 audio file can be identified by its ".mp3" file extension. Similarly, an MP4 video file can be identified by the ".mp4" extension. The file extensions of multimedia files do not provide any kind of surety about the format. Even the format name is merely a notion. If you need to process audio and video content, you need to go beyond file extensions. You need to be familiar with multimedia concepts such as containers, codecs, encoders, and decoders. In this chapter, you will gain some basic information about all that and more.

Containers

Multimedia files such as MP4s or MP3s are just *containers* – containers for some audio and/or video content. An MP4 file is a container for some video content written using the *H.264 codec* and some audio content written using the *AAC codec*. It need not be like that for all MP4 files. Some MP4 files may have their video content written using the *Xvid codec* and the audio content written using the *MP3 codec*. Similarly, AVI, MOV WMV, and 3GP are popular containers for audio/video content. Codecs can differ from file to file even if their extensions are the same. A multimedia file may have the wrong extension because of some human error. You can expect all sorts of combinations in the wild.

When the codecs are not what is usually expected in a container, you may encounter annoying format errors in playback devices. Sometimes, you may be able to fix the error by simply renaming the file with the correct

V. Subhash, *Quick Start Guide to FFmpeg*, https://doi.org/10.1007/978-1-4842-8701-9_3

extension. At other times, you will have to re-encode the file using *codecs* supported by the device. So, what does it mean when a device says it only supports certain "codecs"?

Codecs, Encoders, and Decoders

When audio and video recordings transitioned from analog to digital, equipment manufacturers developed algorithms to store audio waveforms and video frames in a scheme retrievable by computer software. Initially, these storage schemes were proprietary, and their documentation was not publicly available. With the rise in the popularity of digital media devices, interoperability and open standards became necessary.

When multimedia (audio or video) content is written or stored in a computer file, it is written in a specific retrievable format developed by the manufacturer of the multimedia equipment. The algorithm used to read or write multimedia content in a specific format became known as a **codec** (coder-decoder). The software used for writing the content using the codec became known as an **encoder**. The software used to read the written content became known as a **decoder**. A camera uses an *encoder* chip to store captured video. A TV uses a *decoder* chip to play the video from a USB drive. On a personal computer, the logic of encoder and decoder chips is installed as a *software codec*.

Raw audio or video requires a lot of space when stored on a computer file. The multimedia industry, led by camera manufacturers and computer companies, has developed several compression techniques to squeeze multimedia content on to as few bytes of storage as possible. The efficiency of the compression techniques varies. When the compression discards some content (assuming that the human ear or the eye would not miss it) for a dramatic decrease in the size of the file, the technique would be known as *lossy compression*. When no content was discarded, the technique was known as *lossless compression*. Lossless compression techniques are not used everywhere because of the high file-space requirement.

To suit real-world requirements, most codecs provide options to their algorithm so that a balance between file size reduction and detail loss can be specified on a preset or *ad hoc* basis. You will do the same when you use FFmpeg. For example, in the following command, to convert an uncompressed audio from a microphone recording to a lossy compressed audio format, several settings such as bitrate, number of channels, and sampling frequency are specified.

```
ffmpeg -i uncompressed-stereo.wav \
       -c:a libmp3lame -b:a 128k -ac 2 -ar 44100 \
       compressed.mp3
```

☞ You will learn more about these settings in later chapters, but for now just be aware that they are often required.

Demuxers and Muxers

I have been using FFmpeg for years without knowing what demuxers and muxers were. Even now, I cannot care less. Well… maybe a little. A *demuxer* is a software component that can read a multimedia input file so that a decoder can work on it. Similarly, a *muxer* writes data to a multimedia output file after it has been processed by an *encoder*. Between a decoder and encoder, some processing work may be done, or it may even pass directly to the other end. Here is all that you need to know:

- To write to a particular container format, the format's muxer is required.
- To read from a particular container format, a demuxer is required.

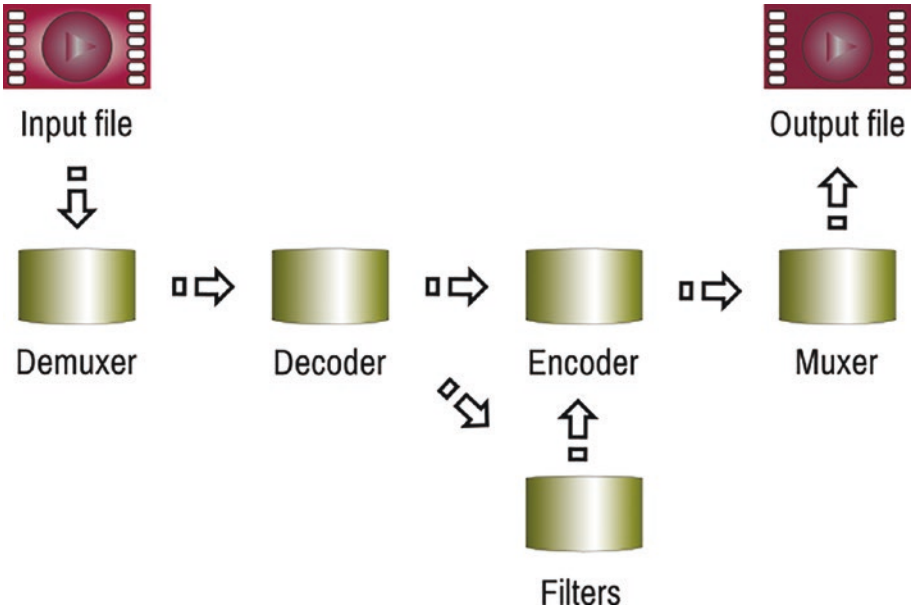


Figure 3-1. *This schematic shows how different components in FFmpeg work together to give the output you want*

For example, to read and write to the MP4 format, an MP4 demuxer and an MP4 muxer are required. FFmpeg automatically takes care of muxers and demuxers so that you do not have to bother with them. However, there may come situations when you do have to explicitly address them.

```
~/Desktop
$ ffmpeg -h demuxer=gif
Demuxer gif [CompuServe Graphics Interchange Format (GIF)]:
GIF demuxer AVOptions:
  -min_delay         <int>        .D.....  minimum valid delay between
  -max_gif_delay     <int>        .D.....  maximum valid delay between
  -default_delay     <int>        .D.....  default delay between frame
  -ignore_loop       <boolean>    .D.....  ignore loop setting (netsca
```

Figure 3-2. *This demuxer help output provides a clue as to how to create endlessly looping GIF animations*

Summary

In this chapter, you learned some theoretical concepts about multimedia formats, containers, and codecs. In the next chapter, we will delve deeper into the container and learn how to refer to its constituents from the command line using index numbers.

CHAPTER 4

Media Containers and FFmpeg Numbering

In the previous chapter, you learned that a multimedia file is actually a container. On the inside, it encloses multimedia *streams* and *metadata*. In this chapter, you will learn what streams and metadata are and how you can access them from the command line. The sections in this chapter are arranged for easy access and completeness. It may not be possible for you to understand all of it on your first read. Return to this chapter a few times to get a full understanding.

Containers

A container can have several streams. A stream could be audio, video, subtitles, or a file attachment.

In an MP4 video file or container, you will usually find a *video stream* and an *audio stream*. In an MP3 file, you will find an audio stream and maybe some IDv3 tags (such as title, album, and artist) as metadata.

If you have one of those rare multi-angle DVDs, then each camera angle will be represented by a separate *video stream*. Multi-language videos will have an *audio stream* for each language. DVD subtitles for

V. Subhash, *Quick Start Guide to FFmpeg*, https://doi.org/10.1007/978-1-4842-8701-9_4

multiple languages are represented as individual *subtitle streams*. MKV files may have custom font files for displaying the subtitles. These font files will be represented as *file-attachment streams*.

In an audio stream, there can be several channels. A mono audio stream has only one channel. A stereo stream has two channels - left and right. A DVD movie's 7.1 surround sound stream has eight channels - front left, front right, center left, center right, rear left, rear right, and one LFE (low frequency effects).

FFmpeg identifies these streams, channels, and metadata **using index numbers** so that you can refer to them from the command line.

Container Internals

Logically, the internals of a multimedia file look like this. A container needs to have at least one stream. Everything else is optional. It is all right for a video file to not have album art, subtitles, custom fonts, or tags (global metadata), but one video stream and one audio stream are usually expected.

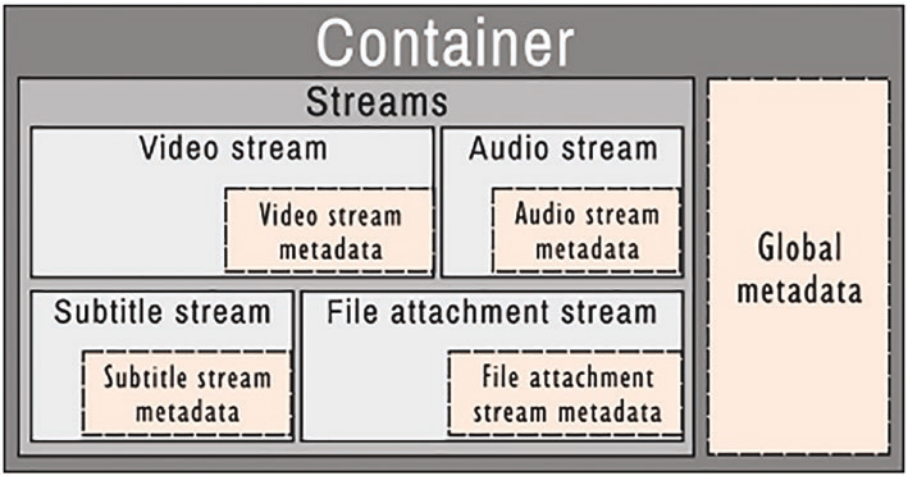


Figure 4-1. *Internals of a multimedia file container*

From this logical representation, you will note that a multimedia file container may have some global metadata and that each stream in the container can have stream-specific metadata too.

You can use `ffprobe` to display these details for any multimedia file.

```
~/Desktop/FasDrive
$ ffprobe "Flyte1 - Manic Monday (Album).mp3"
Input #0, mp3, from 'Flyte1 - Manic Monday (Album).mp3':
  Metadata:
    title           : Manic Monday (Album)
    album           : Greatest Hits
    artist          : The Bangles
    genre           : Pop
    date            : 1990
    comment         : Purchased by V. Subhash from Flipkart - Flyte
  Duration: 00:03:03.72, start: 0.025056, bitrate: 320 kb/s
    Stream #0:0: Audio: mp3, 44100 Hz, stereo, fltp, 320 kb/s
    Metadata:
      encoder         : LAME3.98r
    Stream #0:1: Video: mjpeg (Baseline), yuvj420p(pc, bt470bg/unkno
    Metadata:
      comment         : Cover (front)
```

Figure 4-2. *This is a sample `ffprobe` output for an audio file*

In this `ffprobe` output, the global metadata for the MP3 file shows ID3 tags such as title, album, and artist. It also includes a "comment" metadata that I added after I bought the music. The metadata for the audio stream shows that it was encoded using the LAME encoder by the music vendor. The album art is shown as a video stream but it has only one frame. More importantly, you should note that FFmpeg refers to the input files and streams using **index numbers starting from 0 (zero)**, instead of 1 (one).

Here is another example; this one is for a video file.

```
~/Desktop
$ ffprobe lucas.mkv
Input #0, matroska,webm, from 'lucas.mkv':
  Metadata:
    encoder         : libebml v1.0.0 + libmatroska v1.0.0
    creation_time   : 2020-02-19T20:30:08.000000Z
  Duration: 00:00:20.13, start: 0.000000, bitrate: 507 kb/s
    Stream #0:0: Video: h264 (High), yuv420p(progressive), 320x180 [SAR
1:1 DAR 16:9], 24 fps, 24 tbr, 1k tbn, 48 tbc (default)
    Stream #0:1: Audio: mp3, 44100 Hz, stereo, fltp, 128 kb/s (default)
    Stream #0:2: Subtitle: ass (default)
    Stream #0:3: Attachment: ttf
    Metadata:
      filename        : Florentia.ttf
      mimetype        : application/x-truetype-font
```

Figure 4-3. *This is a sample* `ffprobe` *output for a video file*

What does this output say?

- The MKV file is identified as the first input file (`#0`).
- It has **global metadata** for creation time but none for title, copyright, comments, etc.
- The first stream (`#0:0`) is a **video stream** and requires a H.264 decoder.
- The second stream (`#0:1`) is an **audio stream** and requires an MP3 decoder. The audio is in stereo, that is, it has two channels.
- The third stream (`#0:2`) is a **subtitle stream** and requires a decoder for the Substation Alpha (SSA) format.
- The fourth stream (`#0:3`) is a custom font for displaying the subtitles. It is stored as a **file-attachment stream**.

- The fourth stream also has some **stream-specific metadata** identifying the font file's name and mimetype. This is important because the SSA subtitles may refer to the font by this name.

☞ *Mimetype* is a more rigorous file-type definition (than file extensions) and is usually used by websites to identify downloads to web browsers.

Input and Output Files

An `ffmpeg` command can have multiple input and output files. The following command has two input files and one output file. (For now, ignore the line with the filter. Filters are explained in Chapter 7.)

```
ffmpeg -i solar.mp4 -i overlay.png \
       -filter_complex "overlay=370:260:" \
       watermarked-solar.mp4
```

☞ When specifying multiple input files, place options specific to one input file on the left side of `-i` option. Whatever specified after the file name applies to the next input file (`-i`) or (in its absence) the next output file.

☞ ffmpeg can also read from streams and write to them. The streams can be piped from/to another command and also transported over a network protocol. For more information, read the official documentation on *protocols*.

A video of my solar inverter and the cover image of one of my books are the input files. The command renders the image at 370 pixels from the left edge and 260 pixels from the top edge of the video.

Figure 4-4. *The output video is the input video with the overlaid input image*

The two **input files were specified using the `-i` option**. An MP4 video file is input file `#0` and a PNG image file is input file `#1`. **The output file, as is always, has been specified last**.

```
~/Desktop
$ ffmpeg -i solar.mp4 -i overlay.png \
>        -filter_complex "overlay=370:260:" \
>        watermarked-solar.mp4
Input #0, mov,mp4,m4a,3gp,3g2,mj2, from 'solar.mp4':
  Metadata:
    comment         : MOV00127
  Duration: 00:01:36.93, start: 0.000000, bitrate: 390 kb/s
    Stream #0:0(und): Video: h264 (High) (avc1 / 0x31637661)
SAR 1:1 DAR 4:3], 351 kb/s, 24 fps, 24 tbr, 12288 tbn, 48 tb
    Metadata:
      handler_name    : VideoHandler
    Stream #0:1(und): Audio: aac (LC) (mp4a / 0x6134706D), 8
40 kb/s (default)
    Metadata:
      handler_name    : SoundHandler
Input #1, png_pipe, from 'overlay.png':
  Duration: N/A, bitrate: N/A
    Stream #1:0: Video: png, rgba(pc), 245x200 [SAR 11811:11
br, 25 tbn, 25 tbc
Stream mapping:
  Stream #0:0 (h264) -> overlay:main (graph 0)
  Stream #1:0 (png) -> overlay:overlay (graph 0)
  overlay (graph 0) -> Stream #0:0 (libx264)
  Stream #0:1 -> #0:1 (aac (native) -> aac (native))
Press [q] to stop, [?] for help
```

Figure 4-5. *The output of the command shows the index numbers used for the input files and streams*

The output of the command shows that the first stream in the first input file is a video stream and is numbered `#0:0`. The second stream in that file is an audio stream and is numbered `#0:1`. The first stream in the second input file (the PNG image file) is considered as a video stream even though it has only one (image) frame and is identified as `#1:0`.

You can refer to streams by their type. In the previous command, the streams were as follows:

- `0:v:0` (first file's first video stream) or `0:0` (first file's first stream)
- `0:a:0` (first file's first audio stream) or `0:1` (first file's second stream)
- `1:v:0` (second file's first video stream) or `1:0` (second file's first stream)

For this to become clear, spend some time studying the screenshot in Figure 4-5.

Suppose that a multi-language DVD video file had one video stream and two audio language streams. The streams can be referred as follows:

- `0:v:0` (first video stream) or `0:0` (first stream)
- `0:a:0` (first audio stream) or `0:1` (second stream)
- `0:a:1` (second audio stream) or `0:2` (third stream)

☞ In the output of `ffmpeg` commands, you will encounter index numbers ignoring the stream type. To make your FFmpeg commands somewhat fail-safe, I recommend that you refer to streams by their type instead.

As you may have guessed, the stream-type identifier for video is `v` and `a` for audio. There are others as given in Table 4-1.

Table 4-1. *Stream-type identifiers*

Stream type	Identifier
Audio	a
Video	v
Video (not images)	V
Subtitles	s
File attachments	t
Data	d

After displaying the information about the input files and streams, `ffmpeg` will list how the input streams will be processed and mapped to intermediate and final streams. Then, it will list the final output files and their streams. In a `bash` terminal, you can press the key combination Ctrl+S if you wish to pause and study this information. Otherwise, all of this information will quickly flash past your terminal as `ffmpeg` will then post a huge log of informational, warning, and error messages as it performs the actual processing of the input data.

Maps

With multiple input files, FFmpeg will use an internal logic to choose which input streams will end up in the output file. To override that, you can use the `-map` option. Maps enable you to specify your own selection and order of streams for the output file. You can specify stream mapping in several ways:

```
-map InputFileIndex
all streams in file with specified index

For example, -map 1 means
all streams in second (1) input file.
```

`-map InputFileIndex:StreamIndex`
the stream with specified index in file with specified index

For example, `-map 0:2` means
third (2) stream in first (0) input file.

`-map InputFileIndex:StreamTypeIdentifier`
all streams of specified type in file with specified index

For example, `-map 1:s` means
all subtitle (s) streams in second (1) input file.

`-map InputFileIndex:StreamTypeIdentifier:StreamIndex`
among streams of specified type in file with specified index, the stream with specified index

For example, `-map 2:s:1` means
second (1) subtitle (s) stream in third (2) input file.

Information overload? Let me explain with an example. When I created this stop-motion video a few years ago, I used a gramophone recording as the background music. Typical of old record music, it had a lot of sound artifacts. At that time, I did not know much about FFmpeg. So, I used FFmpeg to extract the audio as an MP3 file but used the free *Audacity* program to apply a *low-pass filter*. Then, I used FFmpeg again to swap the original audio with the MP3 fixed by Audacity.

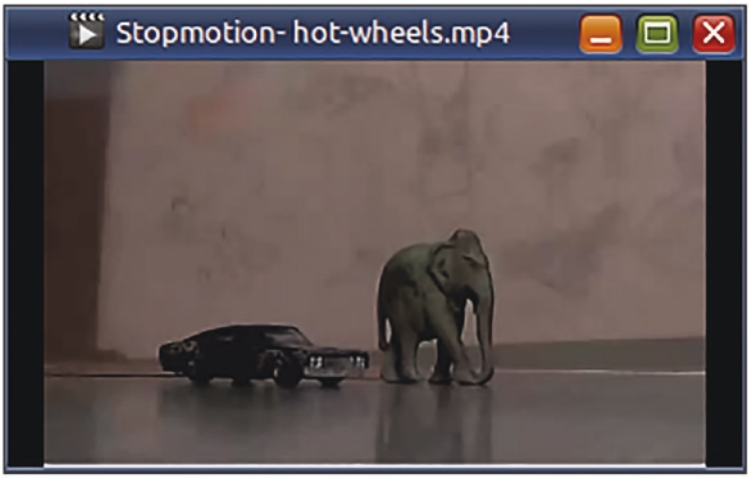


Figure 4-6. *The audio of this video had gramophone sound artifacts*

```
# Extract the audio
ffmpeg -i Stopmotion-hot-wheels.mp4 \
       -map 0:1 \
       Stopmotion-hot-wheels.mp3

# Apply low-pass filter to Stopmotion-hot-wheels.mp3
# using Audacity and export to Stopmotion-hot-wheels-fixed.mp3

# Swap the existing audio track with the mp3 fixed by Audacity
ffmpeg -i Stopmotion-hot-wheels.mp4 \
       -i Stopmotion-hot-wheels-fixed.mp3 \
       -map 0:0 -map 1:0 \
       -codec copy \
       Stopmotion-hot-wheels-fixed.mp4
```

☞ `-codec copy` or `-c copy` copies the streams as they are, instead of unnecessarily re-encoding or converting them again. It saves a lot of time.

In the first command, I included a map for the second stream `(0:1)` in the MP4 file and saved it as an MP3 file. (I assumed that the second stream was an audio stream. It need not be.) I then corrected errors in the MP3 file using Audacity. In the second command, the first input file (the MP4 file) had two streams – `(0:0)` and `(0:1)` – same as in the first command. (More assumptions.) The second input file (the "fixed" MP3) had one stream `(1:0)`. In the second command, I used the first file's first stream `(0:0)` and the second file's first and only stream `(1:0)`. Alternatively, I could have typed the command by mapping to the first file's first video stream `(0:v:0)` and the second file's first audio stream `(1:a:0)`.

```
ffmpeg -i Stopmotion-hot-wheels.mp4 \
       -i Stopmotion-hot-wheels-fixed.mp3 \
       -map 0:v:0 -map 1:a:0 \
       -codec copy \
       Stopmotion-hot-wheels-fixed.mp4
```

☞ This alternative *fail-early* approach is safer, as it can protect you from typing mistakes.

The audio stream in the original MP4 `(0:1)` or `(0:a:0)` gets discarded because it was not included in any of the maps. If I wanted to retain the original audio stream, I can add another map for it as a second audio stream. The fixed audio track will be played by default by media players. I can manually select the second audio track with the remote or a menu option to hear the unfixed original audio.

```
ffmpeg -i Stopmotion-hot-wheels.mp4 \
       -i Stopmotion-hot-wheels-fixed.mp3 \
       -map 0:v:0 -map 1:a:0 -map 0:a:0 \
       -codec copy \
       Stopmotion-hot-wheels-fixed-n-restored.mp4
```

You can use maps when generating multiple output files with one command.

```
ffmpeg -i solar.mp4 \
  -map 0:1 -c:a libmp3lame -b:a 128k solar-high.mp3  \
  -map 0:1 -c:a libmp3lame -b:a 64k solar-low.mp3
```

The `-map` options provide a new set of streams available for options specified after them. Options such as `-codec` or `-ac` will only affect streams specified by the `-map` options before them, not the streams available in the input files.

Metadata

Metadata means data about data. When using FFmpeg, metadata is read by the demuxer and/or written by the muxer. The data is usually specified as **key-value pairs**. For a media file, the metadata can be global (for the entire file) or specific to a stream in the file. Each container format specifies a limited set of metadata keys. The MP3 format, for example, supports metadata keys such as title, artist, album, and copyright. You can specify metadata for individual streams as follows:

```
      -metadata:s:StreamIndex or

      -metadata:s:StreamTypeIdentifier:StreamIndex
```

This command sets metadata at the global/file/container level.

```
ffmpeg -i solar.mp4 -codec copy \
       -metadata title="Me Solar Inverter" \
       solarm.mp4
```

Figure 4-7. *The background video has no metadata, and the video player just displays the file name on the window title. In the foreground video, title metadata is available, and the video player displays that text instead of just the file name*

The `ffprobe` output in Figure 4-8 shows potentially incriminating information about a moonshiner MP3.

```
~/Desktop
$ ffprobe raisa.mp3
Input #0, mp3, from 'raisa.mp3':
  Metadata:
    major_brand     : mp42
    minor_version   : 0
    compatible_brands: isommp42
    title           : Musiki - Мужики
    encoder         :
    artist          : Raisa Prikolnaya - Раиса Прикольная
  Duration: 00:02:50.71, start: 0.000000, bitrate: 136 kb/s
    Stream #0:0: Audio: mp3, 44100 Hz, stereo, s16p, 128 kb/s
    Stream #0:1: Video: png, rgb24, 640x360, 90k tbr, 90k tbn, 90k tbc
    Metadata:
      title           : Screenshot-HD-2015-YouTube-mp4-1.png
      comment         : Other
```

Figure 4-8. *This `ffprobe` output shows that this inveterate pirate had downloaded a music video from Youtube and ripped the audio!*

```
ffmpeg -y -i raisa.mp3 \
       -map 0 -c copy \
       -metadata:s:v:0 title='raisa.png' \
       raisa2.mp3          # Smooth!
```

This command makes no changes to the MP3 except for the value of the incriminating `title` metadata of the album art.

```
    Stream #0:1: Video: png, rgb24, 640x360 [SAR 3780:3780 DAR 16:9],
90k tbr, 90k tbn, 90k tbc
    Metadata:
      title           : raisa.png
      comment         : Other
```

Figure 4-9. *This updated `ffprobe` output shows that the pirate has smoothly changed the metadata. Maybe he was doing researchez academique! Non? Nhyet?*

Remember my stopmotion video with multitrack audio? I can use the `-metadata` option to give its audio streams an informative language name.

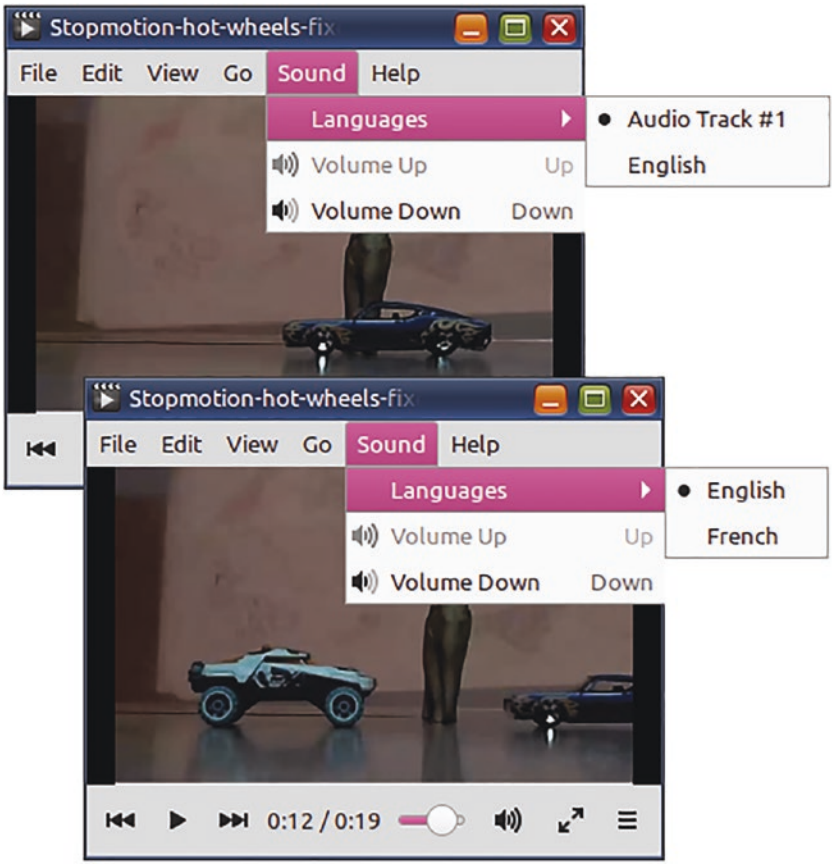


Figure 4-10. *If you do not specify a language name for an audio track, media players may make wrong assumptions*

```
ffmpeg -i Stopmotion-hot-wheels.mp4 \
       -i Stopmotion-hot-wheels-fixed.mp3 \
       -map 0:v:0 -map 1:a:0 -map 0:a:0 \
       -codec copy \
       -metadata:s:a:0 language="eng" \
       -metadata:s:a:1 language="fre" \
       Stopmotion-hot-wheels-fixed-n-restored.mp4
```

Remember that to set the language names for subtitle streams, the `-metadata` option should refer to subtitle streams, not audio streams.

```
-metadata:s:s:0 language="eng" \
-metadata:s:s:1 language="fre" \
```

The `StreamIndex` refers to the index of the stream IN THE OUTPUT FILE. The `s` after `-metadata:` identifies itself as metadata for a stream. Do not mistake it for subtitles. Also, remember that metadata is all about the output file. Do not use any numbering from the input file(s).

☞ Apart from streams (`-metadata:s`), metadata can be specified for DVD chapters (`-metadata:c`) and DVD programs (`-metadata:p`). They are not covered by this book.

☞ You can learn more about metadata in Chapter 10.

Metadata Maps

Have you noticed that when you convert MP3 files, the album art or the meta tags get lost? This is because of improper or no metadata mapping. Metadata can get lost when you convert files or create new files from multiple input files. The `-map_metadata` option helps you correctly route metadata from input files to output files. Its value is specified in a rather twisted manner. The **left is the destination** and the **right is the source**.

```
-map_metadata InputFileIndex:MetadataSpecifier or
-map_metadata:g InputFileIndex:MetadataSpecifier or
-map_metadata:MetadataSpecifier InputFileIndex:⏎
MetadataSpecifier
```

> Where
>
> `MetadataSpecifier` is either `g` or `s:StreamType` (all streams) or `s:StreamType:StreamIndex` (some stream)

Yeah, it made my head spin too! Take your time. Nobody does metadata mapping on their first excursion into FFmpeg. Take the slow lane.

The following example copies global metadata from the second input file (`-map 1`) as the global metadata for the output file. This ensures that the MP3 tags are copied as the video's metadata.

```
ffmpeg -y -i raisa.png -i raisa.mp3  \
       -c:a copy -c:v mjpeg \
       -map 0 -map 1 \
       -map_metadata 1 \
       raisa.mp4
```

The next example copies global metadata from the second input file both globally (`:g`) and to the audio stream (`:s:a`). The global metadata from the second input file can be specified either as `1:g` or simply as `1`. Global output metadata can be typed as `-map_metadata:g` (as below) or simply as `-map_metadata` (as above).

```
ffmpeg -y -i raisa.png -i raisa.mp3  \
       -c:a copy -c:v mjpeg \
       -map 0 -map 1 \
       -map_metadata:g 1:g -map_metadata:s:a 1 \
       raisa.mkv    #Does not work with MP4
```

What is the advantage of this command? If someone decides to extract just the audio stream from the MKV, the metadata does not get omitted. The stream and the MKV (global) both have a copy of the metadata from the MP3 file. The original metadata will survive even in the extracted audio stream.

```
Input #0, matroska,webm, from 'raisa2.mkv':
  Metadata:
    title           : Musiki - Мужики
    MAJOR_BRAND     : mp42
    MINOR_VERSION   : 0
    COMPATIBLE_BRANDS: isommp42
    ARTIST          : Raisa Prikolnaya - Раиса Прикольная
  Duration: 00:02:50.74, start: 0.000000, bitrate: 131 kb/s
    Stream #0:0: Video: mjpeg, yuvj444p, 640x360 [SAR 1:1 DAR 16:9], 25
fps, 25 tbr, 1k tbn, 1k tbc (default)
    Stream #0:1: Audio: mp3, 44100 Hz, stereo, s16p, 128 kb/s (default)
    Metadata:
      title           : Musiki - Мужики
      MAJOR_BRAND     : mp42
      MINOR_VERSION   : 0
      COMPATIBLE_BRANDS: isommp42
      ARTIST          : Raisa Prikolnaya - Раиса Прикольная
```

Figure 4-11. *The global metadata has been duplicated to the audio stream metadata as well*

The `-metadata` option overrides `-map_metadata` mapping.

Channel Maps

Audio streams can have one or more channels. Monaural audio has only one channel. Stereo music has two channels - left and right. DVD movies can have two or six or eight channels for playback on both stereo and surround speaker systems.

To pin down the channels exactly as you want in the output file, you need to use the `-map_channel` option. It can be specified as follows:

```
        -map_channel
        InputFileIndex.StreamIndex.ChannelIndex
```

or as

`-map_channel -1`

if you want the channel muted.

The `-map_channel` options specify the input audio channels and the order in which they are placed in the output file.

Imagine that the audio channels in an MP4 file are mixed up. When you wear headphones, in either ear, the voices are heard for people on the opposite side in the video. You can fix it by the following:

```
ffmpeg -i wrong-channels.mp4 \
       -c:v copy \
       -map_channel 0.1.1 -map_channel 0.1.0 \
       fine-channels.mp4
```

In a stereo audio stream, the channel order is `0.1.0` (left) followed by `0.1.1` (right). When you use a channel map of `0.1.1` followed by `0.1.0`, the channels get switched.

For the next example, imagine that you are using headphones in a work environment. You want to have one ear for music and one ear for surroundings. You could mute one of the channels.

```
ffmpeg -i moosic.mp3 \
       -map_channel 0.0.0 -map_channel -1 \
       moosic4lefty.mp3
```

☞ No, you should not make it mono. Mono audio will be heard on both sides.

In some videos, the left and right audio channels are independent tracks. What these content creators do is place the original audio on one channel and the most annoying royalty-free music on the other. Instead

of deleting the offending channel, you could move each channel to a separate audio stream while preserving the original stereo stream in a third stream.

```
ffmpeg -y -i zombie.mp4 \
       -map 0:0 -map 0:1 -map 0:1 -map 0:1 \
       -map_channel 0.1.0:0.1 -map_channel 0.1.1:0.2 \
       -c:v copy \
       zombie-tracks.mp4
```

The first stream in the output file will be the original video (`0.0`). The left channel (`0.1.0`) will be the second stream (`0.1`). The right channel (`0.1.1`) will be the third stream (`0.2`). The original stereo audio will become the fourth stream. (Yes, the second and third streams will be mono audio.)

What about the numbers after the colon? That is explained by the full definition for channel maps:

```
       -map_channel InputFileIndex.InputFileStreamIndex.⏎
       ChannelIndex:OutputFileIndex.OutputFileStreamIndex
```

How do you like them apples? The second part beginning with the colon is optional. It is for placing the mapped input audio channel on a specified output stream.

☞ Channel mapping numbers use dots, not colons. The colon is used only when you begin to specify the output stream.

☞ Channel mapping cannot be used to mix channels from multiple input files.

☞ When you make changes to the channels, the audio will be converted again and this takes time. It will not be done quickly like with `-c:a copy`.

Do Not Use the `-map_channel` Option

The `-map_channel` option, with its difficulties, is on its way out. The FFmpeg version 5.1 (released in July 2022) shows this warning.

```
The -map_channel option is deprecated and will be removed.
It can be replaced by the 'pan' filter, or in some cases by
combinations of 'channelsplit', 'channelmap', 'amerge' filters.
```

With newer `ffmpeg` versions, the previous commands can be rewritten using filters, which you will learn in a later chapter.

```
# Switch right and left channels of stereo audio
ffmpeg -i wrong-channels.mp4 \
       -c:v copy \
       -filter_complex "channelmap=map=FR-FL|FL-FR" \
       fine-channels.mp4

# Silence right channel
ffmpeg -i moosic.mp3  \
       -c:v copy \
       -filter_complex "pan=stereo|FL=FL|FR=0" \
       moosic4lefty.mp3

# Split channels to separate audio streams
# and also preserve existing audio stream
```

```
ffmpeg -y -ss 0:0:20 -t 0:0:20 -i zombie.mp4 \
       -c:v copy \
       -filter_complex "channelsplit[L][R]" \
       -map 0:v:0 -map '[L]' -map '[R]' -map 0:a:0 \
       -codec:a:0 aac -ac:a:0 1 \
       -codec:a:1 aac -ac:a:1 1 \
       -codec:a:2 copy \
       zombie-tracks.mp4
```

☞ The `-codec` and `-ac` options are limited to streams specified by the `-map` options specified before them.

Summary

In this chapter, you learned about how to access streams and metadata. You also learned how to pick and choose what streams and metadata you would like to have in the output file(s).

As mentioned in the beginning of this chapter, it is not necessary that you grasp every detail in this chapter on the first go. As you read forthcoming chapters, certain things mentioned in this chapter will become clearer. If not, you can always return to this chapter.

CHAPTER 5

Format Conversion

The main reason that so many people use `ffmpeg` is its amazing ability to convert files from one format to another. `ffmpeg` supports so many formats that I doubt there is any competition even from paid software. In this chapter, you will learn how to perform these conversions and customize them to extract the best quality from the source files.

No-Brainer Conversions

The default output format in many Linux multimedia programs is OGV and OGG files. Sadly, very few consumer electronic devices support these two formats. I use `gtk-recordMyDesktop` to screen capture my computer demos, and it creates OGV video files. Before I can play the files on my TV, I need to convert them to MP4 format.

```
ffmpeg -i video1.ogv video1.mp4
```

An Ogg ringtone will play fine on an Android phone but not on a feature phone, which usually only supports MP3 and MIDI ringtones. Converting Ogg to MP3 is easy with FFmpeg.

```
ffmpeg -i alarm.ogg alarm.mp3
```

FFmpeg can guess the output format based on the file extension you have used for the output file. It will automatically apply some good preset conversion settings (defaults). You can specify custom conversion settings too.

V. Subhash, *Quick Start Guide to FFmpeg*, https://doi.org/10.1007/978-1-4842-8701-9_5

Conversion Options

Table 5-1 lists a few FFmpeg options that are useful when converting files. You will learn how to use them in the rest of this chapter.

Table 5-1. *Some FFmpeg conversion options*

Option	For
-y	Prevent prompting before overwriting any existing output file
-b:a	Set audio bitrate
-c:a	Specify audio encoder or decoder
-ar	Set audio sampling rate
-ac	Set number of audio channels
-b:v	Set video bitrate
-c:v	Specify video encoder or decoder
-r	Set video frame rate
-pass	Specify number of the encoding pass
-passlogfile	Specify prefix for multi-pass encoding log files
-f	Force specified format (or oss, alsa, rawvideo, concat, image2, null…)
-shortest	Stop all processing when any one output stream is completely processed
-vn	Do not process video
-an	Do not process audio
-sn	Do not process subtitles

Obsolete/Incorrect Options

FFmpeg is fault-tolerant to an extent but do not be sloppy in typing the options. You should avoid using `-r:a` instead of `-ar` (audio sampling rate). Instead of conventions such as `-acodec` and `-vcodec`, you should be using `-c:a` or `-c:v` instead. Support for such old practices may be removed in future.

Codec Option

The `-codec` option is used to specify an encoder (when used before an output file). When used before an input file, it refers to the decoder. (`ffmpeg` may have more than one decoder and encoder for a particular codec.) Choose the correct name from the output of the command `ffmpeg -encoders` or `ffmpeg -decoders`, and not from that of `ffmpeg -codecs`.

The `-codec` option can also be specified for all streams for a particular type, such as `-codec:a` for all audio streams or `-codec:s` for all subtitle streams or for a particular stream using its index. For each stream, only the last applicable `-codec` option will be considered. If you use the value `copy` for the encoder, `ffmpeg` will copy applicable streams as is without using an encoder.

How do you know which codec (encoder name) you need to use for a particular format? For an MP3 file, you could try the following:

```
ffmpeg -encoders | grep mp3
```

It may not be so straightforward with other formats. Browse through the full output of the command `ffmpeg -encoders` to become familiar with codec names. Sample output of this command is available in Annexure 3. Then, you will learn that H.264 and MPEG-4 codecs have something to do with MP4 files. You could also use `ffprobe` on existing file samples and find prospective codec names.

```
~/Desktop
$ ffmpeg -encoders | grep mp3
 A..... libmp3lame           libmp3lame MP3 (MPEG audio layer 3) (code
 A..... libshine             libshine MP3 (MPEG audio layer 3) (codec

~/Desktop
$ ffmpeg -encoders | grep mp4

~/Desktop
$ ffmpeg -encoders | grep mpeg4
 V.S... mpeg4                MPEG-4 part 2
 V..... libxvid              libxvidcore MPEG-4 part 2 (codec mpeg4)
 V..... mpeg4_v412m2m        V4L2 mem2mem MPEG4 encoder wrapper (codec
 V..... msmpeg4v2            MPEG-4 part 2 Microsoft variant version 2
 V..... msmpeg4              MPEG-4 part 2 Microsoft variant version 3

~/Desktop
$ ffmpeg -encoders | grep h26
 V..... h261                 H.261
 V..... h263                 H.263 / H.263-1996
 V..... h263_v412m2m         V4L2 mem2mem H.263 encoder wrapper (codec
 V.S... h263p                H.263+ / H.263-1998 / H.263 version 2
 V..... libx264              libx264 H.264 / AVC / MPEG-4 AVC / MPEG-4
 V..... libx264rgb           libx264 H.264 / AVC / MPEG-4 AVC / MPEG-4
 V..... h264_nvenc           NVIDIA NVENC H.264 encoder (codec h264)
```

Figure 5-1. *`ffmpeg` lists a lots of encoders, several pages full. You may miss some important ones if you make assumptions and filter the output. Use the command `ffmpeg -encoders | more` to conveniently browse the full output*

Sample Conversion with Custom Settings

If I wanted to convert a HD video downloaded from the Internet for playing on my old portable media player, I would use these settings.

```
ffmpeg -i net-video.mp4 \
       -s 320x240 \
       -c:v mpeg4 -b:v 200K -r 24 \
```

```
        -c:a libmp3lame -b:a 96K -ac 2  \
        portable-video.mp4
```

The output video stream uses MPEG4 codec with qvga (320x240) dimensions, 200K bitrate, and a 24 frames-per-second rate. The output audio stream uses MP3 codec (Lame encoder) with two-channel audio (stereo) and 96K bitrate.

☞ You will know what values to use for each setting only if you make it a habit to use `ffprobe` on new types of files that you encounter.

☞ The bitrate is how densely the audio or video content is stored in the container. The greater the compression, the lesser is the bitrate and file size, and so is the quality. You need to find a balance between quality loss and file size reduction.

Multi-pass Conversion

In multi-pass encoding, `ffmpeg` processes the video stream multiple times to ensure the output video is close to the specified bitrate. `ffmpeg` creates a log file for each pass. In the initial passes, the audio is not processed and video output is not saved (dumped on null device). In the final pass, however, you will have to specify the audio conversion settings and the output file. In the next example, the conversion from the previous section is performed using two passes.

This is the first pass.

```
ffmpeg -y -i net-video.mp4 \
         -s 320x240 -c:v mpeg4 -b:v 194k -r 24 \
```

```
        -f mp4 -pass 1 -passlogfile /tmp/ffmpeg-log-
        net-video \
        -an /dev/null
```

☞ Windows users should use `NUL` instead of `/dev/null`.

And, this is the last pass.

```
ffmpeg -y -i net-video.mp4 \
        -s 320x240 -c:v mpeg4 -b:v 194k -r 24 \
        -pass 2 -passlogfile /tmp/ffmpeg-log-net-video \
        -c:a libmp3lame -ac 2 -b:a 96K \
        portable-video.mp4
```

Multiple passes of the first kind may be required for achieving a particular bitrate. Use the same video conversion settings for all passes.

☞ When the streams meet the specified bitrates, you will also know exactly how big the file will be. Just multiply the bitrate with the duration of the video. The reverse is also true. You can target a particular file size (allowing for some deviation) by specifying a proportional bitrate for both the audio and video. Conversion with constant bitrate was popular when DVD videos were encoded (ripped off) to fit on a CD.

Conversion for Maximum Compression and Quality

Multimedia codecs provide a trade-off between speed, quality, and compression. Now that we have almost unlimited online and offline space, constant quality rather than constant bitrate is preferred. With

the H.264 codec, you can achieve the required quality and compression in *one pass* using the `-crf` (CRF or Constant Rate Factor) option and by specifying a processing "preset." The `-crf` option affects quality.

```
x264 Presets    x264 Tune    x264 Profiles
~~~~~~~~~~~~    ~~~~~~~~~    ~~~~~~~~~~~~~
ultrafast       film         baseline
superfast       animation    main
veryfast        grain        high
faster          stillimage
fast            psnr
medium          ssim
slow            fastdecode
slower          zerolatency
veryslow
placebo
```

Figure 5-2. *This extract from the output of an old script shows preset and tuning variables supported by the H.264 encoder*

```
ffmpeg -i solar.mp4 \
       -c:v libx264 -crf 21 -preset fast \
       -c:a copy \
       solar-CONVERTED.mp4
```

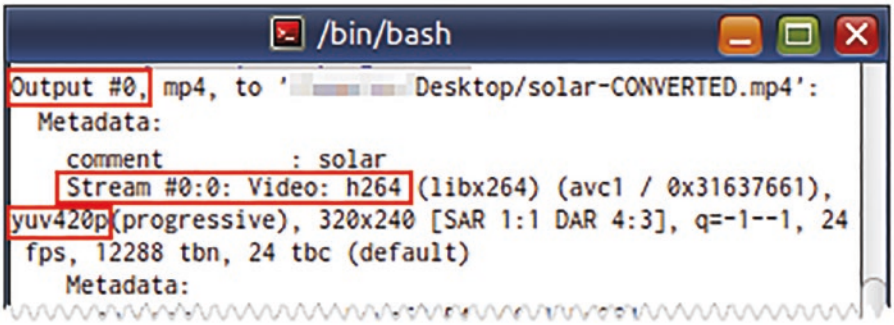


Figure 5-3. *The ffmpeg output stream details will tell you which pixel format has been used*

The CRF range is from 0 (lossless) to 63 (worst) for 10-bit pixel formats (such as `yuv420p10le`) and 0 to 51 for 8-bit pixel formats (such as `yuv420p`). You can determine the pixel format from the `ffmpeg` output of a similar file conversion. The median can be 21 for 8-bit encoder and 31 for 10-bit encoder.

What the heck is a pixel format? All that you need to know about pixel format (at this stage) is that it is a data-encoding scheme used to specify the colors of each pixel (dots) in a video frame. FFmpeg supports these pixel formats: `monob, rgb555be, rgb555le, rgb565be, rgb565le, rgb24, bgr24, 0rgb, bgr0, 0bgr, rgb0, bgr48be, uyvy422, yuva444p, yuva444p16le, yuv444p, yuv422p16, yuv422p10, yuv444p10, yuv420p, nv12, yuyv422,` and `gray`.

In addition to the processing preset, you can also specify a `-tune` option depending on the kind of video that you have selected. The values `psnr` and `ssim` are used to generate video quality metrics and are not normally used in production. `zerolatency` output can be used for streaming. `fastdecode` can be used for devices that do not have a lot of processing power. `grain` is to prevent the encoder from being confused by grainy videos.

Audio Conversion

This command uses the Lame MP3 encoder to convert an Ogg audio file to a 128K-bitrate two-channel (stereo) MP3 file.

```
ffmpeg -i alarm.ogg \
       -c:a libmp3lame \
       -ac 2 \
       -b:a 128K \
       alarm.mp3
```

☞ There is a better method for converting to MP3 files. You will find it in Chapter 11.

Audio Extraction

Some video files have great sound. Music videos are good examples. How do you extract their audio? Well, drop the video stream and copy the audio stream to an audio file.

```
# Matroska audio
ffmpeg -i music-video.mp4 -c:a copy music-video.mka
# MPEG4 audio - FFmpeg flounders
ffmpeg -i music-video.mp4 -vn -c:a copy music-video.m4a
```

☞ Without -vn, the video stream will get copied to the m4a file! Hurray for redundant options! Le paranoid survive!

Matroska audio or ".mka" files support several audio codecs. The ".m4a" files support AAC (MPEG4 audio) codec.

If you already know that the audio stream in the MP4 file has been encoded with MP3 codec (as they do sometimes), you can `-codec:a copy` the audio stream to a ".mp3" file. Most of the time, however, you will have to *encode* it to MP3. Files with extension ".mka" and ".m4a" are not supported by many playback devices. The following command converts the audio stream of the video file using the Lame encoder to create a two-channel (stereo) MP3 file encoded at 128K bitrate.

```
ffmpeg -i music-video.mp4 \
       -c:a libmp3lame -b:a 128K -ac 2 \
       music-video.mp3
```

You can simultaneously output audio in different bitrates using multiple `-map` options.

```
ffmpeg -i music-video.mp4 \
  -vn \
  -map 0:a -c:a libmp3lame -b:a 128K music-high.mp3 \
  -map 0:a -c:a libmp3lame -b:a 64K music-low.mp3
```

☞ As one understands, this is strictly for limited doomsday archival purposes…. Several films and music records have been lost to studio fires. Anything can happen. Cite the 2020 pandemic. 🤓

Extract Stills from a Video (Video-to-Image Conversion)

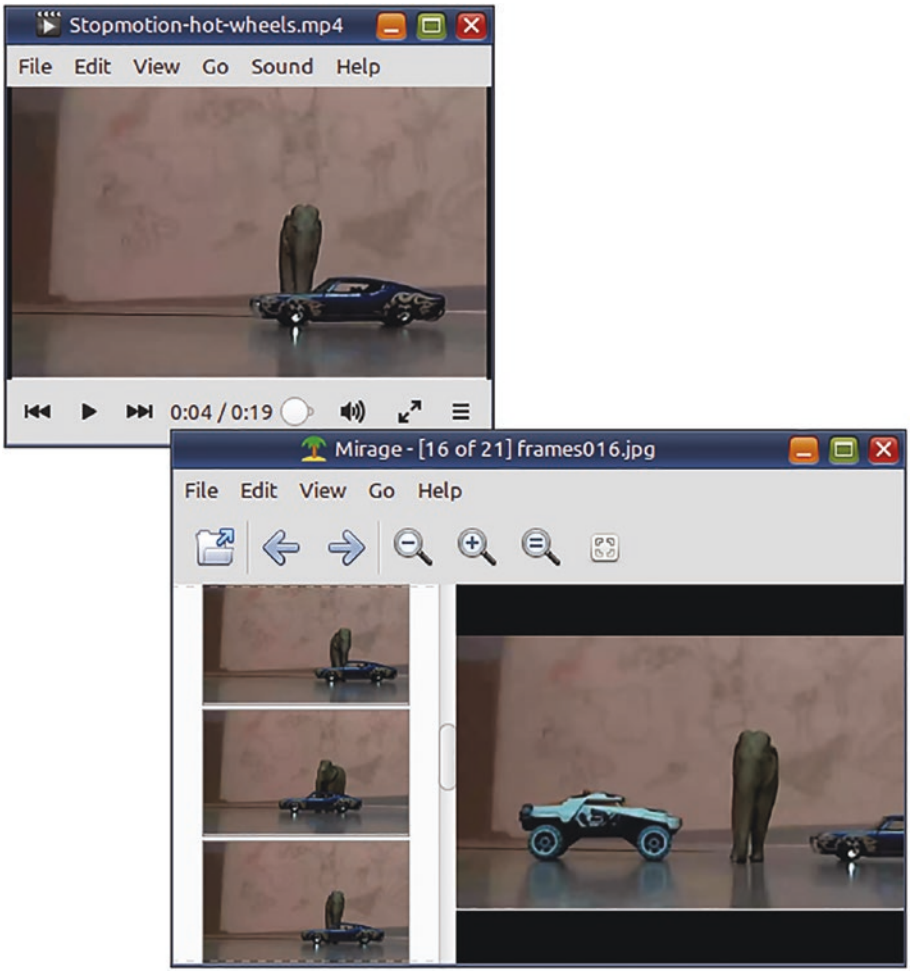


Figure 5-4. *A video and the still-image frames extracted from it*

To extract video frames as image files, you need to use the `-f image2` option. The numbering of the output images is specified in the name of the output file. The format mask of the output file is similar to that of the `printf` function in the C programming language. In the mask used in the next command, `%` is for character output, `0` is for padding with zeros instead of spaces, `3` is for the total number of digits, and `d` is for integer numbers.

```
# Extract images at the rate of 1 frame per second from
the video
ffmpeg -y -i Stopmotion-hot-wheels.mp4 \
       -r 1 \
       -f image2 \
       frames%03d.jpg 2> /dev/null
```

☞ Most videos are encoded with a frame rate of 24, 25, 30, or even 60 frames per second. Be careful with your extraction rate and length of the video, or you will quickly run out of space.

Use the `-r` option to restrict the number of images generated for each second of the source video. You can omit the `-r` option to extract all frames (and let it be determined by the frame rate of the source video) but

- Use small video clips as the source
- Use `-t` and `-ss` options (described in Chapter 6) to restrict the extracted duration of the source video

Image-Conversion Settings

Table 5-2 lists some FFmpeg conversion options that are useful when working with image files. Although this book will describe how to use them, more comprehensive information will be found in the official FFmpeg documentation.

Table 5-2. *ffmpeg image-conversion options and examples*

Option	Purpose
`-f image2`	Force conversion to and from images
`-f image2pipe`	Force image conversion for output piped over to another command
`-loop 1`	Repeat the processing of the input image indefinitely
`-pix_fmt yuv420p`	Use yuv420p pixel format when converting to image formats

Create Video from Images (Image-to-Video Conversion)

FFmpeg can also do the reverse by creating a video from several images (when they are numbered serially). The duration of the video depends on the number of images available and frame rate you have specified.
If the `-r` option in the video-to-image conversion was higher (in the previous command), say between 12 and 30, a lot more images would have been extracted, and this video would have been smoother.

```
ffmpeg -r 1 -i frames%03d.jpg \
       -s qvga -pix_fmt yuv420p \
       Stopmotion-hot-wheels-reconstituted.mp4 2> /dev/null

ffplay -autoexit \
       Stopmotion-hot-wheels-reconstituted.mp4 2> /dev/null
```

☞ All input images should be of the same format and dimensions.

☞ The `-pix_fmt yuv420p` option is necessary to ensure such unusual video files play all right in most media player devices.

Create a Slideshow from Several Images

In the previous section, the output video ran out quickly because there were not many input images. If you want each input image to appear for longer than a second, then you need to specify a `-framerate` option for them as well. An input frame rate of 1/3 ensures that a frame plays for 3 seconds.

```
ffmpeg -y -framerate 1/3 -i image%02d.jpg \
          -filter:v \
             "scale=eval=frame:w=640:h=480:
              force_original_aspect_ratio=decrease,
              pad=640:480:(ow-iw)/2:(oh-ih)/2:yellow" \
          -pix_fmt yuv420p -r 24 \
          slide.mp4
```

☞ You will learn more about filters in Chapter 7.

The preceding command also takes care of images with irregular dimensions and ensures that they are resized appropriately.

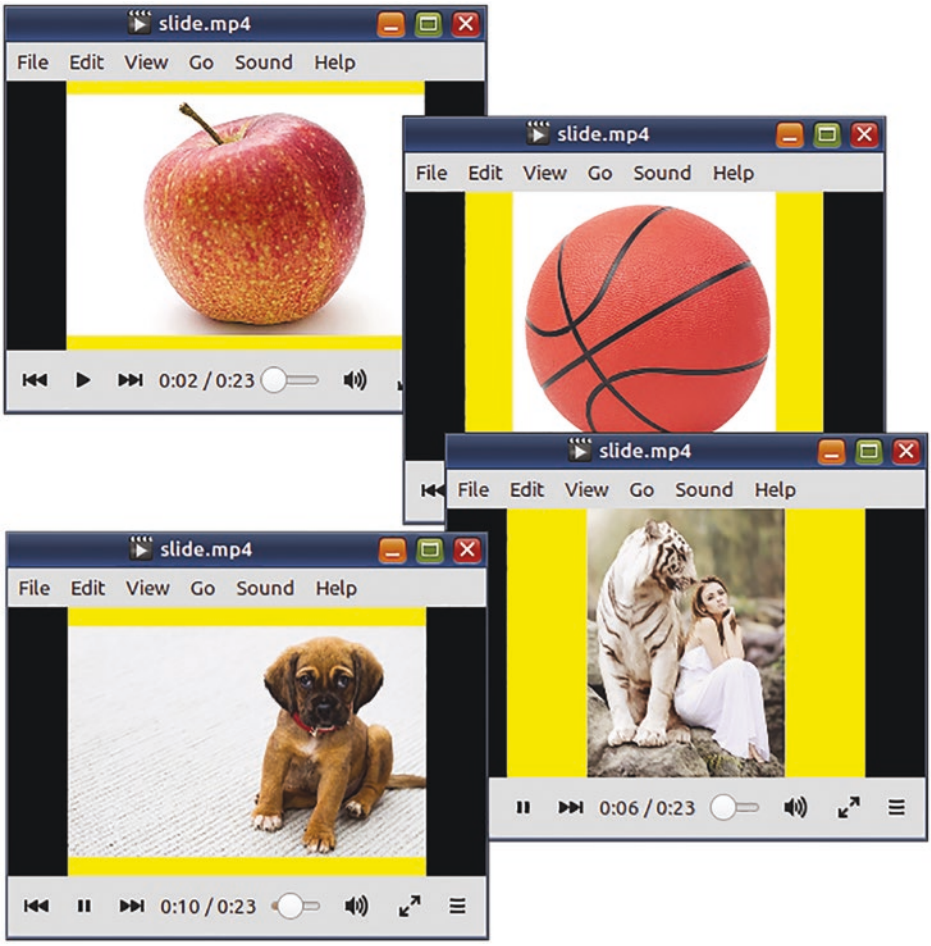


Figure 5-5. *This video was created from several disproportionate images*

When you have input images in no particular naming sequence, then you can pipe them like this:

```
cat *.png | \
     ffmpeg -y -f image2pipe \
            -framerate 1/3 -i - \
            -filter:v \
```

```
              "scale=eval=frame:w=640:h=360:
               force_original_aspect_ratio=decrease,
               pad=640:360:(ow-iw)/2:(oh-ih)/2:black" \
            -c:v libx264 -r 24 -s nhd -pix_fmt yuv420p \
            slide2.mp4
```

Create a GIF from a Video

The ancient GIF format supports only 256 colors. You need to use `palettegen` and `paletteuse` filters to downsample the source video to this limited number of colors.

```
ffmpeg -y  -i bw.m4v \
       -filter_complex \
         "fps=7,scale=w=320:h=-1:flags=lanczos,split[v1][v2];
          [v1]palettegen=stats_mode=diff[p];
          [v2][p]paletteuse=dither=bayer:bayer_scale=4" \
         bw-4.gif
```

You need to experiment a lot with the filters to understand what will work and what will not. A set of values that do well to optimize the file size for one source video may do poorly for another video. GIF optimization is extremely unpredictable. Learn more from this article:

```
https://engineering.giphy.com/how-to-make-
gifs-with-ffmpeg/
```

In an experiment with the production of a GIF file from a video, I found that

- With a `bayer_scale` of 0 (with the `dither=bayer` mode), the animation is smooth but suffers from the appearance of a dotted texture. The file size is on the higher side.

- When moving to the highest value of 5 (default is 2), the frames are clearer but start to suffer from intermittent banding. The file size is smaller.

The results may be quite different for another video file.

If you are stuck with an older version of FFmpeg that does not have the `palettegen` and `paletteuse` filters, you can make FFmpeg output the frames to ImageMagick (`convert` or `magick`). (The hyphens in the following command refer to standard output and input.)

```
ffmpeg -y -i bw.m4v \
       -filter:v "fps=10,scale=w=320:h=-1:flags=lanczos" \
       -c:v ppm \
       -f image2pipe - | \
  convert -delay 10 -  \
          -loop 0 \
          -layers optimize \
          bw.gif
```

APNG

A better alternative to GIF animations is APNG. This format has limited support from image-viewing and image-editing applications but has near-universal support from desktop and mobile web browsers. Like PNG and unlike GIF, APNG supports millions of colours. This means that its colours will not have to be downsampled and will be very close to those in the source content. APNG animation files are typically bigger than animated GIFs.

If you are converting GIF animations to APNGs, then ImageMagick is the tool you should use, not `ffmpeg`.

```
magick animated.gif animated.apng
```

The image frames in a GIF will already be downsampled to 256 colours. To create a richer animated PNG, try to use the source frames in PNG format.

```
magick -delay 200 -loop 0 \
       chapter-image-*.png \
       -units PixelsPerInch -density 72 -resize '>x300' \
       animation-unlikely-stories.apng
```

If you are converting a video to APNG, then you can use `ffmpeg`.

```
ffmpeg -i bw.m4v \
       -vf "scale=w=250:h=-2, hqdn3d, fps=6" \
       -dpi 72 -plays 0 \
       bw.apng
```

In this command, `-dpi` is an APNG encoder option and `-plays` is an APNG muxer option. The *high-quality denoise 3d* filter reduces blemishes introduced by the scaling filter. Learn more about these options from the official documentation or by typing:

```
ffmpeg -help muxer=apng
ffmpeg -help encoder=png
ffmpeg -help filter=hqdn3d
```

Create a Video Using an Image and an MP3

How do you play an MP3 in a media player that will only play MP4 files? Find a thumbnail or album art for the MP3 and churn it out as a video. The following command uses an image as a video stream encoded with MJPEG codec.

```
ffmpeg -i Blobfish_face.jpg -i blobfish.mp3 \
       -c:v mjpeg -c:a copy \
       -map 0:v:0 -map 1:a:0 \
       -disposition:v:0 attached_pic \
       "Weird Fins - 17 - The Blobfish.mp4"
```

This command generates only one image frame in the MP4. The image frame is not encoded as a regular video stream for the entire duration of the audio.

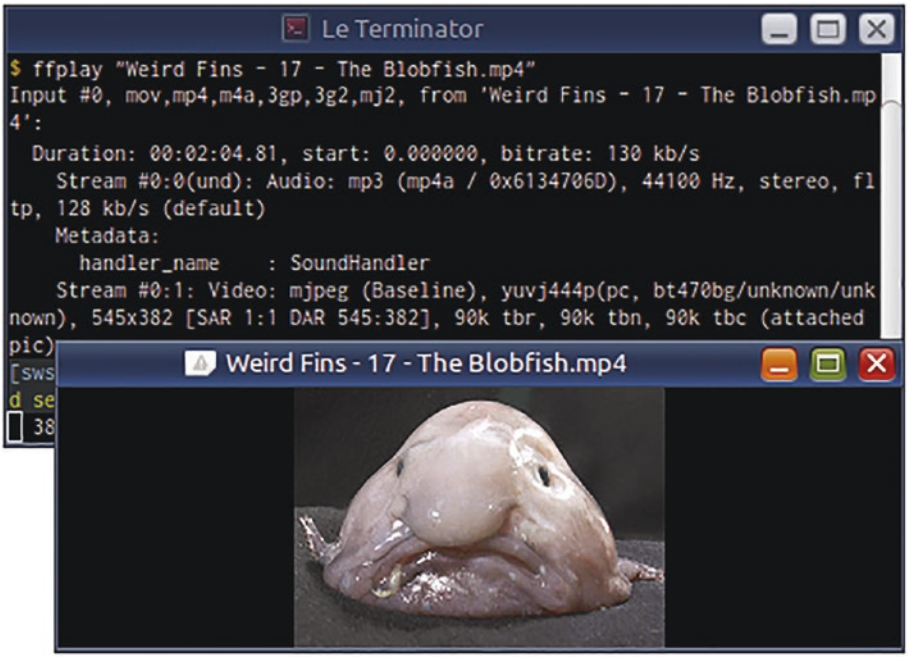


Figure 5-6. *This video does not really have any video, just one frame from an image*

However, not all media players will accept this trickery. On my computer, Totem media player does not show the image at all and plays it like a regular audio file. VLC displays the image because it uses FFmpeg internally. If your player shirks its duty, you will have to encode the image for the full duration of the audio.

```
ffmpeg -y -i  blobfish.mp3 \
  -loop 1 -framerate 12 -i Blobfish_face.jpg \
  -shortest -s qvga -c:a copy \
  -c:v libx264 -pix_fmt yuv420p \
  "Weird Fins - 17 - The Blobfish (no tricks).mp4"
# The album art image loops forever so the
# podcast audio creates the shortest output stream
```

☞ This MP3 was part of 18 MP3 files of the "Weird Fins" podcast published by the US NOAA. It got lost and buried when they redesigned their site. Some years ago, I recovered these files, tagged them and uploaded them to Archive.org.

Convert Online Videos to Audio

YouTube-DL is an open source command-line program that can download online videos for offline use. It supports several online video sites. Many journalists use it to grab still images for their articles about Internet videos. However, the entertainment industry has decided to challenge the legal status of this utility. The Electronic Frontier Foundation (EFF) and surprisingly GitHub (owned by Microsoft) have come up with a defense initiative for its survival.

```
        https://youtube-dl.org
```

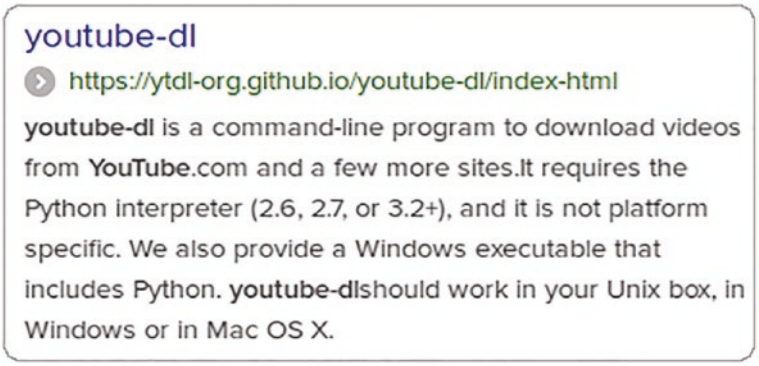


Figure 5-7. *This is a description of* `youtube-dl` *published by a search engine*

Assuming that your `~/bin` directory is in the `$PATH` environment variable, you can install `youtube-dl` locally using the following:

```
wget https://yt-dl.org/downloads/latest/youtube-dl \
     -O ~/bin/youtube-dl

chmod +x ~/bin/youtube-dl

youtube-dl --version
```

☞ YouTube-DL will run from anywhere. You do not have to install the file to a privileged location like the site says.

☞ If `youtube-dl` does not run, maybe Python 3 is not in the `PATH`. Start it with `/usr/bin/python3 youtube-dl ...`.

You can make `youtube-dl` use `ffmpeg` to convert the downloaded files. Many audio podcasts are posted to online video sites. To only listen to them in the Audacious media player, I use a command like this:

```
youtube-dl -f 140 -x  \
           --audio-format mp3  \
           --exec 'audacious {} & ' \
           https://www.youtube.com/watch?v=yypDkqAErx0
```

`youtube-dl` will not only download and convert the audio (from AAC) to MP3 (using `ffmpeg`), but it will also launch a command when the conversion process is complete. That command can be for your media player. `youtube-dl` will replace `{}` in the command string with the name of the output (MP3) file.

Convert Text to Audio

If your `ffmpeg` executable has been built-in with support for the `libflite` text-to-speech synthesizer library, then you can convert text content to spoken words.

```
ffmpeg -f lavfi \
       -i "flite=textfile=speech.txt:voice=slt" \
       speech.mp3
```

This speech filter supports several voices. On my computer, it lists `awb`, `kal`, `kal16`, `rms`, and `slt` as available voices. Unfortunately, the female voice sounds a bit dopey.

```
ffprobe -f lavfi "flite=list_voices=1"
```

I like the male-only `espeak` utility better. The defaults are good, and you can change several settings.

Conversion Settings for Specific Storage Medium

If you use the -target option, certain conversion settings appropriate for the specified storage option will be applied. The values for this option can be `vcd`, `svcd`, `dvd`, `dv`, and `dv50`. They can be prefixed with `ntsc`, `pal`, or `film` for more specific targets. For the actual settings used by these targets, refer the official FFmpeg documentation.

```
ffmpeg -i movie.avi -target ntsc-dvd movie.mpg
```

☞ VCD (MPEG-1), DVD (MPEG-2), and DV (digital tape) are very old targets and consume more space than MPEG-4.

☞ If you want to extract still images from movies, optical media is usually the best source.

Summary

In this chapter, you learned how to convert multimedia content in the form of audio, video, image, and text. You also learned to customize conversion settings to suit different formats, coder/decoders, and mediums. In the next chapter, you will learn how to edit videos using `ffmpeg`.

CHAPTER 6

Editing Videos

I used to save DVDs as ISO files (whole-DVD backups) so that I could play them on my media player box. Each ISO took up several gigabytes (GBs) on my hard disk that I eventually ran out of space. Now, I use FFmpeg and store DVDs as MP4s of around just one GB.

While FFmpeg makes it very easy to convert multimedia files, as you learned in the previous chapter, storing them in their entity is not always feasible or required. Sometimes, you need just a few clips, not the whole shebang. You may want to combine parts of one video with parts of other videos. You can also downsize the videos to conserve space. In `ffmpeg` terms, you want to cut, concatenate, and resize videos. In this chapter, you will learn to do just that.

Resize a Video

You can resize a video using the `-s` option. The dimension of a video is usually specified as *Width***x***Height*. That is an "x" as in "x-mas" in the middle. When editing or converting videos, you will have to specify the video dimension using this syntax. The next command resizes a VGA-size (640x480) video to a VCD-size (352x288) video.

```
ffmpeg -i dialup.mp4 \
       -s 352x288 \
       dialup.mpg
```


V. Subhash, *Quick Start Guide to FFmpeg*, https://doi.org/10.1007/978-1-4842-8701-9_6

FFmpeg supports certain easy-to-remember literals that you can use in place of the actual numbers for the `-s` option. They are listed in Table 6-1.

Table 6-1. *FFmpeg option and values for setting the dimensions of a video*

Option	For					
-s	Video dimensions (literal or actual)					
	Literal	**Dimensions**	**Literal**	**Dimensions**	**Literal**	**Dimensions**
	ntsc	720x480	uxga	1600x1200	hd1080	1920x1080
	pal	720x576	qxga	2048x1536	2k	2048x1080
	qntsc	352x240	sxga	1280x1024	2kflat	1998x1080
	qpal	352x288	qsxga	2560x2048	2kscope	2048x858
	sntsc	640x480	hsxga	5120x4096	4k	4096x2160
	spal	768x576	wvga	852x480	4kflat	3996x2160
	film	352x240	wxga	1366x768	4kscope	4096x1716
	ntsc-film	352x240	wsxga	1600x1024	nhd	640x360
	sqcif	128x96	wuxga	1920x1200	hqvga	240x160
	qcif	176x144	woxga	2560x1600	wqvga	400x240
	cif	352x288	wqsxga	3200x2048	fwqvga	432x240
	4cif	704x576	wquxga	3840x2400	hvga	480x320
	16cif	1408x1152	whsxga	6400x4096	qhd	960x540
	qqvga	160x120	whuxga	7680x4800	2kdci	2048x1080
	qvga	320x240	cga	320x200	4kdci	4096x2160
	vga	640x480	ega	640x350	uhd2160	3840x2160
	svga	800x600	hd480	852x480	uhd4320	7680x4320
	xga	1024x768	hd720	1280x720		

A video's horizontal dimension divided by the vertical dimension is sometimes referred to as the **aspect ratio**. This is further influenced by the dimensions of individual pixels that make up the video. (Remember that a video frame is a matrix of dots or pixels in lines and columns.) This pixel-level aspect ratio is known as the **sample aspect ratio (SAR)**.

When a video is resized, `ffmpeg` (or whichever video authoring software that is used) would have automatically adjusted the pixel dimensions (or the SAR) from square to rectangular so that the video will be played with the proper aspect ratio.

If you want a video to be played at a particular aspect ratio, you need to set the **display aspect ratio (DAR)**. This value is calculated from the width-and-height ratio multiplied by the SAR. If for some reason, the SAR value is not present in the video, it is assumed to be 1. If this makes the video distorted, set the desired DAR using the `setdar` filter and let `ffmpeg` figure out the internal SAR.

```
ffmpeg -i "distorted.mpg" \
         -vf setdar=dar=4/3 \
         restored.mpg
```


You will learn more about filters in Chapter 7.

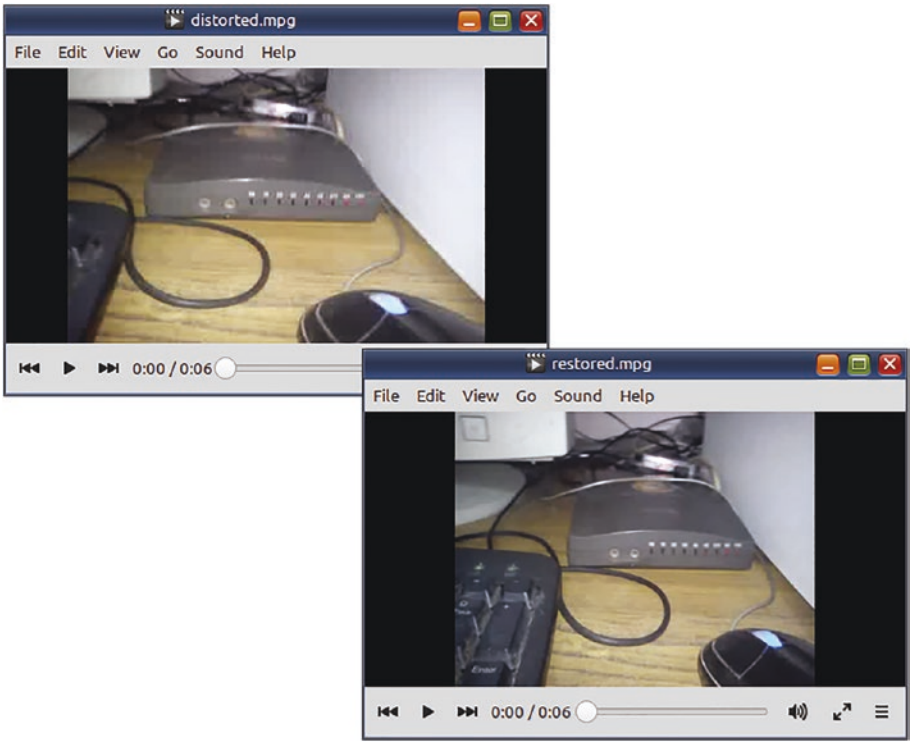


Figure 6-1. *The distortion in the background video was fixed using a filter that changed the DAR (display aspect ratio)*

These ratios may seem similar but there are subtle differences, as presented in Table 6-2.

Table 6-2. *Terms related to video dimensions*

Term	Description
Aspect ratio	= video width ÷ video height ——————————— *Standard definition ratio is 4:3. For widescreen, it is 16:9*
Sample aspect ratio (SAR) a.k.a **pixel aspect ratio**	= pixel width ÷ pixel height ——————————— *For square pixels, it is 1. For rectangular pixels, it will be a fraction*
Display aspect ratio (DAR)	= (video width ÷ video height) × sample aspect ratio or = video aspect ratio × sample aspect ratio

Editing Options

Some often used video- and audio-editing options are listed in Table 6-3.

Table 6-3. *More ffmpeg options for editing*

Option	For
-t	Duration (in hh:mm:ss[.xxx] format or in seconds) of the output file
-ss	Timestamp of playback location (in hh:mm:ss[.xxx] format or in seconds) from which processing needs to be performed
-c:v, -c:a, -c:s	Use specified encoder (not codec) for specific type of stream ——————————— If you use the value `copy` as in `-c copy`, `ffmpeg` will not use an encoder and just copy the stream(s)

Cut a Portion of a Video

If the video segment that you want to remove is the beginning, then use the `-ss` option to specify the timestamp from which the content needs to be copied.

```
ffmpeg -ss 00:01:00 -i sponsored-video.mp4 \
       the-video.mp4
```

☞ Use the `-ss` option before the `-i` option so that `ffmpeg` can quickly jump to the location of the specified timestamp. If you place it after the input file and before the output file, there will be a delay as `ffmpeg` decodes all the data from the beginning to the timestamp and then discards it (as it is not wanted)!

The timestamp values can be specified in the format `hh:mm:ss.ms`. Parts that are zero in the beginning can be omitted, as shown in Table 6-4.

Table 6-4. *Examples of time or duration values*

Usage	Implication
20	20 seconds
1:20	One minute and 20 seconds
02:01:20	Two hours, 1 minute, and 20 seconds
02:01:20.220	Two hours, 1 minute, 20 seconds, and 220 milliseconds
20.020	20 seconds and 20 milliseconds

☞ Before the milliseconds value, there needs to be a dot, not a colon.

If the video segment that you want to remove is the ending, then use `-t` option to specify the duration of the content that needs to be copied from the beginning.

```
ffmpeg -i long-tail.mp4 \
       -t 00:01:00 \
       no-monkey.mp4
```

If you want to cut from the middle, then you need to use both options.

```
ffmpeg -ss 00:01:00 -i side-burns.mp4 \
       -t 00:1:10 \
       clean-shaved.mp4
```

In this case, `ffmpeg` starts cutting `-t` duration of content from the timestamp specified by the `-ss` option, not from the beginning.

All of these commands will re-encode the video. Because the (raw) source video (from which the input video was created) is not being used, the output video will have lesser quality and have freshly introduced blemishes and artifacts.

You may encounter another problem here. When you do not specify conversion settings, then FFmpeg will use its own default encoder settings. If your uncut video had better quality than encoder defaults, then you may end up with lesser quality. If the input file had lower quality, then the encoder defaults may result in increased file size.

To avoid such problems, run `ffprobe` on the input file and use similar conversion settings with `ffmpeg`.

```
~/Desktop
$ ffprobe lucas.mp4
Input #0, mov,mp4,m4a,3gp,3g2,mj2, from 'lucas.mp4':
  Duration: 00:00:20.02, start: 0.000000, bitrate: 575 kb/s
    Stream #0:0(und): Video: h264 (Constrained Baseline) (avc1 / 0x3163
661), yuv420p, 640x360 [SAR 1:1 DAR 16:9], 472 kb/s, 29.97 fps, 29.97 t
r, 30k tbn, 59.94 tbc
    Metadata:
      handler_name    : VideoHandler
    Stream #0:1(eng): Audio: aac (mp4a / 0x6134706D), 44100 Hz, stereo,
fltp, 96 kb/s
    Metadata:
      handler_name    : SoundHandler

~/Desktop
$ ffmpeg -i lucas.mp4 \
>        -vcodec libx264 -b:v 472k -r:v 30 \
>        -acodec libfaac -b:a 96k \
>        -t 0:0:10 \
>        lucas-cut.mp4
```

Figure 6-2. *The ffprobe output shows settings that you can use for the next ffmpeg task*

Cut Without Re-encoding

Apart from losing quality, re-encoding takes time. Cutting without re-encoding does not have these disadvantages. Use the option `-codec copy` to ensure there is no re-encoding and the original quality is retained.

```
ffmpeg -ss 00:01:00 -i dog-eared.mp4 \
       -t 00:1:10 \
       -codec copy \
       clean-cut.mp4
```

There are disadvantages with this option too. The entirety of the audio and video information may not be present at the timestamps you have specified for FFmpeg to make a clean cut. A few seconds of the video may have to be sacrificed or go out of sync. Out-of-sync audio by one or two seconds is not really a problem in videos where the speaker remains in the background.

Use `-codec copy` only when the container of the output file supports the existing codec of the input stream you are trying to copy. You cannot copy streams from an OGV file to a MP4 file, but you can do that with an MKV output file. First, check whether input codecs are among the default codecs listed by the muxer of the output container.

```
ffmpeg -help muxer=matroska | head -5 ; \
ffmpeg -help muxer=ogv | head -5; \
ffmpeg -help muxer=avi | head -5 ; \
ffmpeg -help muxer=mp4 | head -5
```

These commands list the default extensions and codecs used by some popular containers.

```
Muxer matroska [Matroska]:
    Common extensions: mkv.
    Mime type: video/x-matroska.
    Default video codec: h264.
    Default audio codec: vorbis.
Muxer ogv [Ogg Video]:
    Common extensions: ogv.
    Mime type: video/ogg.
    Default video codec: theora.
    Default audio codec: vorbis.
Muxer avi [AVI (Audio Video Interleaved)]:
    Common extensions: avi.
    Mime type: video/x-msvideo.
    Default video codec: mpeg4.
    Default audio codec: mp3.
Muxer mp4 [MP4 (MPEG-4 Part 14)]:
    Common extensions: mp4.
    Mime type: video/mp4.
    Default video codec: h264.
    Default audio codec: aac.
```

Append Videos (Concatenate)

If you need to put together several videos to create one big video containing all of them, then you can use the `concat` demuxer. To use it, you need to first create a text file containing file names or full pathnames of the input videos. The file details should be formatted like this:

- One line should be used for each input file.
- The relative or absolute pathname of a file should be wrapped in quotation marks and preceded by the word "file."

```
file '/tmp/video.mp4'
file '/home/yourname/Desktop/video1.mp4'
file '/media/USB1/DCIM/DS00002.mp4'
```

Ideally, the file locations should be relative to the current directory and have simple file names. Because these files do not satisfy that condition, I have used the option `-safe 0` in this `ffmpeg` command. The next command will re-encode the preceding input files using the specified MP4 settings.

```
ffmpeg -f concat \
       -safe 0 \
       -i list.txt \
       -c:v libx264 -r 24 -b:v 266k -s qvga \
       -c:a libmp3lame -r:a 44000 -b:a 64k -ac 2 \
       mixology.mp4
```

☞ The default for the `-safe` option is `1`. In production environments, it prevents rogue users from using files that would otherwise crash FFmpeg-based software systems.

☞ Use the `-f concat` option setting before the `-i` option.

I advise against the use `-f concat` demuxer. The output files have a tendency to confuse and crash media players. If input videos are not of the same type, the concatenation will fail or the output file will not be playable. The same thing can happen if some of the input files are `-codec copy` veterans. You are lucky if conversion starts at all. If you are forced to use the `concat` demuxer, then read about it in the official documentation. The text file supports other directives (not just `file`) to make it more informative to the demuxer.

For more resilient concatenations, use the `concat` filter as described in Chapter 7.

```
ffmpeg -i engine.mp4 -i coach.mp4 \
    -filter_complex \
      "[0:v:0][0:a:0][1:v:0][1:a:0]concat=n=2:v=1:a=1[v][a]" \
    -map '[v]' -map '[a]' \
    -c:v libx264 -r 24 -b:v 266k -s qvga \
    -c:a libmp3lame -b:a 64k -ac 2 \
    -f mp4 \
    train.mp4
```

Whether you use -codec copy or the concat filter, all the input files should be of the same type (same dimensions, codecs, frame rates, etc.).

Don't Knock -codec copy

After spending considerable time with FFmpeg, you will realize that a lot of multimedia software generate audio/video files that seem to play fine but have a lot of internal encoding errors. Strangely enough, FFmpeg's notorious `-codec copy` option fixes a good many of these container errors.

```
ffmpeg -i smugly.mp4 -codec copy smooth.mp4
```

Summary

FFmpeg provides some very neat options to edit multimedia files from the command line. With some files, you may be able to `-codec copy` the streams. With others, you will have to re-encode them. Both methods have advantages and disadvantages.

In the next chapter, you will finally learn about the `ffmpeg` filters that I have been all along teasing you with.

CHAPTER 7

Using FFmpeg Filters

In the previous chapters, you would have encountered several filters. A great deal of FFmpeg functionality is hidden in them. Most users avoid filters or use them sparingly because the online examples of filters tend to be cryptic. There is a method to the madness. You can crack it. In this chapter, you will learn what filters are and how to use them.

Filter Construction

In an `ffmpeg` command, a filter is used to perform advanced processing on the multimedia and metadata data decoded from the input file(s). A **simple filter** consumes an input stream, processes it, and generates an output stream. The input and output will be of the same type. An **audio filter** (used with the option `-filter:a` or `-af`) consumes an audio stream and outputs an audio stream. A **video filter** (specified by a `filter:v` or `-vf` option) consumes a video stream and outputs a video stream.

You can daisy-chain multiple simple filters to create a **filter chain**. In such a filter chain, the output of one filter is consumed by a subsequent filter. Thus, as a whole, the filter chain will also have one input and one output.

When such a linear filter chain is not possible, you need to use a **complex filtergraph** (with the option `-filter_complex`). A complex filtergraph can contain several filters or filter chains. The constituent filters can have zero to several inputs. They can consume streams of different types and output streams of different types. The number of inputs need not match the number of outputs. It is not necessary for a filter to consume the output of the previous filter.

V. Subhash, *Quick Start Guide to FFmpeg*, https://doi.org/10.1007/978-1-4842-8701-9_7

Some filters known as **source filters** do not have inputs. There are also **sink filters** that do not generate any outputs.

In an `ffmpeg` command, you specify a filter in this fashion:

```
[input label1][input label2]...[input_labelN]filter=
key1=value1:key2=value2...keyN=valueN[output label1]
[output label2]...[output labelN];
```

You need to follow these rules when using filters:

- When a filter is expected to create an output stream, label it with a name in square brackets ([ ]).
- Use these labeled output streams as inputs for other filters or use them in `-map` options. `ffmpeg` automatically names the unlabeled input of the first filter as `[in]` and the unlabeled output of the last filter as `[out]`.
- Between two filters that are part of a linear filter chain (when you daisy-chain them), use a comma (,) as a delimiter. This implies that the output of the first filter is to be consumed as input by the second filter.
- Between two filters that are part of a nonlinear complex filtergraph, use a semicolon (;) as a delimiter. Specify the inputs and outputs using stream identifiers or labels for each filter. If you do not specify input streams, `ffmpeg` will select streams using an internal logic. (Read the official FFmpeg documentation about it.) If the selected input stream cannot be used by the filter, `ffmpeg` will encounter an error. Similarly, when you do not label the output streams, `ffmpeg` will attempt to dump them in the next output file. If the container of the next output file does not support those output streams, `ffmpeg` will encounter an error.

- Specify filter-specific options as key-value pairs. You need to use a colon (:) as a delimiter between them. You can omit the option names (keys) and only use values if you specify them in the same order as specified in the official FFmpeg documentation or help output. This cryptic style is error-prone, difficult to understand, and therefore not recommended.

☞ There are lots of filters and you need to pore over pages of documentation to find the one that will work for you.

Filter Errors

Sometimes, you will encounter a "No such filter" error. This is probably because (out of habit) you placed a semicolon after the last filter. Some filters have an exact number of inputs or outputs. If you fail to identify one of them, `ffmpeg` will throw an error. Other common filter errors are caused when a labeled input or output is not consumed. If you use an output label more than once, you will get an 'Invalid stream specifier' error. An output stream can only be labelled once and used once. If you want to use a filter output stream as input for more than one filter, use the `split` or `asplit` filters to duplicate the stream.

Filter-Based Timeline Editing

Many filters support a generic `enable` option. It can be used to specify the start and end timestamps when the filter should be applied. For example, the option `enable='between(t, 6, 12)'` would ensure that the filter is applied on the video between 6th and 12th seconds of the audio or video.

☞ In the output for `ffmpeg -filters` command, the filters with the flag "T" support timeline editing.

Expressions in FFmpeg Filter Definitions

In the values of some filter options, you can specify algebraic expressions that combine explicit numbers, functions, and some *constants*. (The last two are listed in Table 7-1.) The section *Expression Evaluation* in the documentation describes several functions that can be used in the expressions. FFmpeg defines three *constants* that can be used in any filter.

Table 7-1. *Functions and constants used in ffmpeg filter expressions*

Functions			
abs(x)	floor(expr)	log(x)	sin(x)
acos(x)	gauss(x)	lt(x, y)	sinh(x)
asin(x)	gcd(x, y)	lte(x, y)	sqrt(expr)
atan(x)	gt(x, y)	max(x, y)	squish(x)
atan2(x, y)	gte(x, y)	min(x, y)	st(var, expr)
between(x, min, max)	hypot(x, y)	mod(x, y)	tan(x)
bitand(x, y)	if(x, y)	not(expr)	tanh(x)
bitor(x, y)	if(x, y, z)	pow(x, y)	taylor(expr, x)
ceil(expr)	ifnot(x, y)	print(t)	taylor(expr, x, id)
clip(x, min, max)	ifnot(x, y, z)	print(t, l)	time(0)
cos(x)	isinf(x)	random(x)	trunc(expr)
cosh(x)	isnan(x)	root(expr, max)	while(cond, expr)
eq(x, y)	ld(var)	round(expr)	
exp(x)	lerp(x, y, z)	sgn(x)	
Constants			
PI (22/7)	E (Euler's number or exp(1) ~ 2.718)	PHI (golden ratio or (1+sqrt(5))/2 ~ 1.618)	QP2LAMBDA 118

Several filters define their own *constants*. These are actually real-time variables whose values can change depending on the input files, the processing options, or even time. You need to look at the documentation for each filter to see what these filter constants represent.

☞ You should try to become proficient in the use of filter expressions. They are force multipliers.

☞ When you specify a filter within double quotes (" "), the commas separating the parameters of a function will have to be escaped as `\,` to prevent `ffmpeg` from interpreting them as delimiters used to separate two filters.

Inset Video (Picture-in-Picture Overlay)

Sometimes, people in news media need to use a sign-language inset video. The following `ffmpeg` command scales down a video containing the sign-language track and positions it over the right corner of a news report.

```
ffmpeg -y -i Delphine.mp4 -i accessibility.mp4  \
  -filter_complex \
     "[1:v]scale=w=150:h=150[inset];
      [0:v][inset]overlay=x=W-w-20:y=20[v]" \
  -map '[v]' -map 0:a:0 \
  Delphine-with-accessibility.mp4
```

This command may also be written without the names and only the values of the filter options.

```
ffmpeg -y -i Delphine.mp4 -i accessibility.mp4  \
  -filter_complex \
     "[1:v]scale=150:150[inset];
      [0:v][inset]overlay=W-w-20:20[v]" \
  -map '[v]' -map 0:a:0 \
  Delphine-with-accessibility.mp4
```

☞ If you encounter such commands, they will seem very cryptic. You will have to look up the filter in the official documentation or the help output (`ffmpeg -help filter=scale`) and ascertain the order of the used filter options.

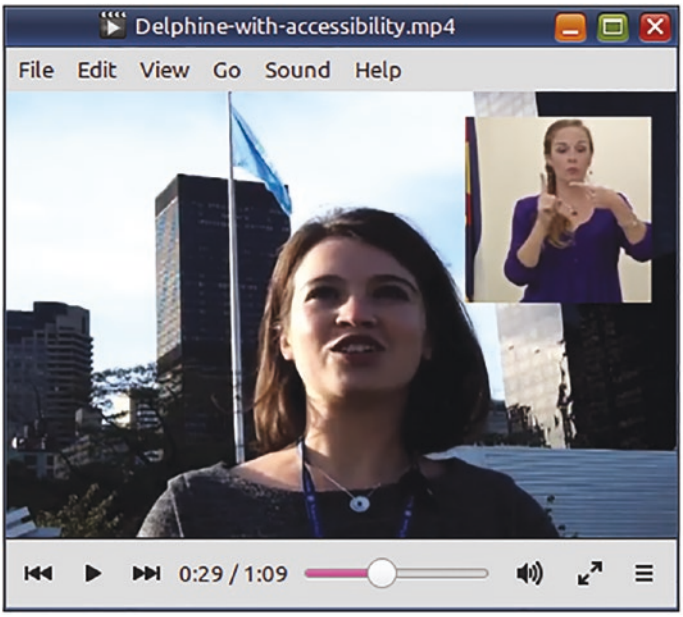


Figure 7-1. *The `overlay` filter has been used to place the sign-language video track in the top-right corner of a news report video*

The `scale` filter specifies actual width and height values (`150:150`) to which the inset video needs to be resized. The `overlay` filter specifies x- and y-coordinates of the top-left corner of the inset video on the news report video. The x-coordinate uses a *filter expression* (`W-w-20`) with *filter constants* `W` (width of the background video) and `w` (width of the inset video) to correctly inset the video 20 pixels away from the right edge of the background video. The y-coordinate is specified with the actual value, that is, 20 pixels from the top edge.

The input for the `scale` filter is the inset video (`[1:v]` or the video stream of the second input file). Its output is labeled `[inset]`. The inputs for the `overlay` filter are the news report (`[0:v]` or the video stream of the first file) and the output of the `scale` filter labeled previously as `[inset]`. The `overlay` filter has one output and it is labeled `[v]`. This overlaid video and the original audio of the news report (`0:a:0`) are then mapped into the output file.

To construct a *filter expression* with useful *filter constants*, you need to refer to the documentation of the filter. If these expressions try to hurt your brain (they will initially), you can specify explicit values. The preceding command can be rewritten as follows:

```
ffmpeg -y -i Delphine.mp4 -i accessibility.mp4  \
  -filter_complex \
     "[1:v]scale=150:150[inset];
      [0:v][inset]overlay=370:20[v]" \
  -map '[v]' -map 0:a:0 \
  Delphine-with-accessibility.mp4
```

Split Video (Side-by-Side Overlay)

When you place two videos side-by-side, their heights should be the same. If you place them one above the other, their widths should be the same. Else, there will be some empty space in the final video.

The sign-language video in the previous section is a 332×332-pixel video. It is smaller than the news report video. If we want them placed side-by-side, the news report video's height needs to be reduced to the height of the sign-language video.

This `scale` filter in this `ffmpeg` command does that. To maintain the same aspect ratio (width ÷ height) of the scaled video, the new width is specified using the filter expression `332*iw/ih`. (The value -2 would have

worked as well. As to how it would, **R**efer **T**he **F**ine **M**anual. ☺) This multiplies the aspect ratio with the new height. (`iw` and `ih` represent filter constants for the width and height of the input video.)

```
ffmpeg -y -i Delphine.mp4 -i accessibility.mp4 \
       -filter_complex "[0:v]scale=332*iw/ih:332[sv];
                        [sv]pad=(iw+332):332:0:0[frame];
                        [frame][1:v]overlay=W-w:0[v]" \
       -map '[v]' -map 0:a:0 \
       Delphine-et-accessibility.mp4
```

Figure 7-2. *The `scale` filter was used to reduce the height of the first video. The `pad` filter has been used to expand the frame of the scaled video. The `overlay` filter has been used to place the second video in the empty area of the expanded frame*

☞ Because the second video is a sign-language video, I discarded its audio. If it were needed, I would have mixed the two audio streams or assigned them to the left and right speaker channels, as described in Chapter 8.

After the `scale` filter, the frame size of the scaled video is expanded sideways so that the second video can be placed in the new empty area. The `pad` filter uses the expression `iw+332` to arrive at the new expanded size of the frame. It then places the scaled video at the top-left corner (`0:0`) of the new frame. That is, the scaled video will be on the left side of the expanded frame.

In the empty area on the right side of the expanded frame (`[frame]`), we place the second input file (`[1:v]`) using the `overlay` filter.

Without using filter expression, the last `ffmpeg` command can be rewritten with actual values as follows:

```
ffmpeg -y -i Delphine.mp4 -i accessibility.mp4 \
       -filter_complex "[0:v]scale=498:332[sv];
                        [sv]pad=830:332:0:0[frame];
                        [frame][1:v]overlay=498:0[v]" \
       -map '[v]' -map 0:a:0 \
       Delphine-et-accessibility.mp4
```

☞ When you want to use the same command on another set of files with different dimensions, you will have to recalculate and re-specify the values. Filter expressions can eliminate a lot of this hassle so use them when you can.

If you do not want the news video to be downscaled, then you could put some white space… (in this case) yellow space around the second video. In the next command, filter expressions and actual values have been used to correctly position the second video in the middle of the expanded frame.

```
ffmpeg -y -i Delphine.mp4 -i accessibility.mp4 \
  -filter_complex
    "[0:v:0]pad=w=(iw+360):h=ih:x=0:y=0:color=yellow[frame];
     [frame][1:v:0]overlay=x=W-360+(360-w)/2:y=(H-h)/2[v]" \
  -map '[v]' -map 0:a:0 \
  -t 0:0:12 -pix_fmt yuv420p \
  Delphine-et-accessibility-et-margin.mp4
```

☞ It is much more easier and faster to use the filters `hstack` and `vstack`. However, these filters require the input videos to have the same pixel format (data encoding scheme of pixel color) and the same dimensions (height for `hstack` and width for `vstack`.)

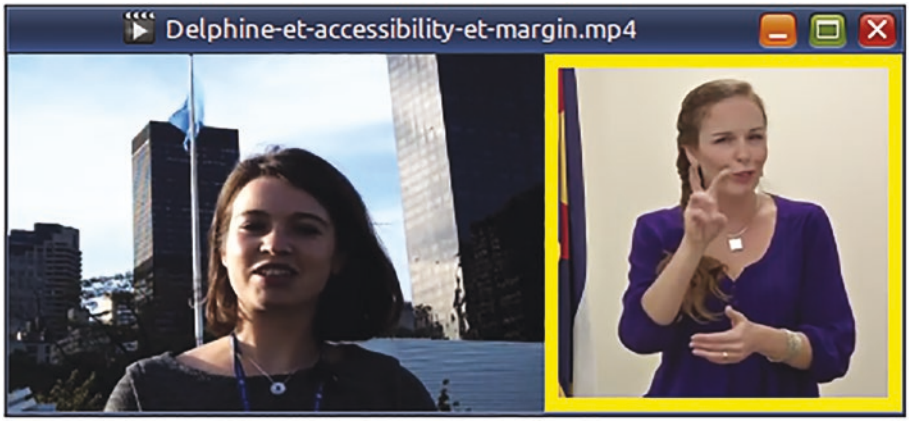


Figure 7-3. *The pad filter was used to expand the width of original frame by 360 pixels while maintaining the same height. The expanded area was given a yellow background that was 360×360 pixels. Using filter expressions with the* `overlay` *filter, the 332×332-pixel second video was placed right in the middle of the yellow background*

Append Videos Using a Filter

In Chapter 6, you learned to concatenate several videos using the `concat` demuxer. The `concat` filter provides more control if you have only a few input files.

```
ffmpeg -i engine.mp4 -i coach.mp4 \
  -filter_complex \
    '[0:v:0][0:a:0][1:v:0][1:a:0]concat=n=2:v=1:a=1[vo][ao]' \
  -map '[vo]' -map '[ao]' \
  -c:v libx264 -r 24 -b:v 266k -s qvga \
  -f mp4 train.mp4
```

This will re-encode the input files, as will any other filter.

Specify the video and audio streams of the input clips or segments in the order that they need to be appended by the filter. `[0:v:0][0:a:0]` refers to the video and audio streams of the first input clip. `[1:v:0]` `[1:a:0]` refers to the video and audio streams of the second clip. The filter option `n` refers to the number of input clips. `v` refers to the number of output video streams, and `a` refers to the number of output audio streams. The concatenated video and audio streams are the filter outputs labeled as `[vo]` and `[ao]`. These labeled outputs are then mapped to the output file.

Delete a Portion of a Video in the Middle

Sometimes, you need to delete part of a video. For that, you can use the `trim`, `atrim`, and `concat` filters. In this command, the second scene (between seconds 16 and 36) is deleted by eliminating it using `trim` and `atrim` filters.

```
ffmpeg -y -i barbara.mp4 \
  -filter_complex \
    "[0:v:0]trim=start=0:end=16, setpts=PTS-STARTPTS[lv];
     [0:v:0]trim=start=36:end=44, setpts=PTS-STARTPTS[rv];
     [0:a:0]atrim=start=0:end=16, asetpts=PTS-STARTPTS[la];
     [0:a:0]atrim=start=36:end=44, asetpts=PTS-STARTPTS[ra];
     [lv][rv]concat=n=2:v=1:a=0[v];
     [la][ra]concat=n=2:v=0:a=1[a]" \
  -map '[v]' -map '[a]' barb-cut.mp4
```

☞ I have used seconds instead of timestamps because the "hh:mm:ss" format requires a lot of nonintuitive escaping.

The `concat` filter is prone to timestamp errors. The `setpts` and `asetpts` filters may be able to fix them. A filter setting with `asetpts=N/SAMPLE_RATE/TB` will generate new timestamps by counting actual samples in the processed audio segments, but it can be used only with constant frame rate videos. A better value is to use `PTS-STARTPTS` (similar to the video filter), as it will also remove empty regions in the audio.

Rotate a Video

Some videos that people take from a mobile phone are rotated by 90 or 180 degrees from normal. You can manually fix them by specifying a `transpose` filter.

```
# Rotate to right
ffmpeg -i slt.mp4 \
       -filter:v "transpose=1" \
       slt-rotated-1.mp4
```

```
# Rotate to left
ffmpeg -i slt.mp4 \
       -filter:v "transpose=2" \
       slt-rotated-2.mp4

# Rotate to left and flip vertically
ffmpeg -i slt.mp4 \
       -filter:v "transpose=0" \
       slt-rotated-0.mp4

# Rotate to right and flip vertically
ffmpeg -i slt.mp4 \
       -filter:v "transpose=3" \
       slt-rotated-3.mp4
```

For the `transpose` filter option `dir`, a value of 1 or 2 turns the video 90 degrees **right or left**. Values 0 and 3 turn the video **left or right** and also vertically flip them. Mobile phone users should stick with the first two values.

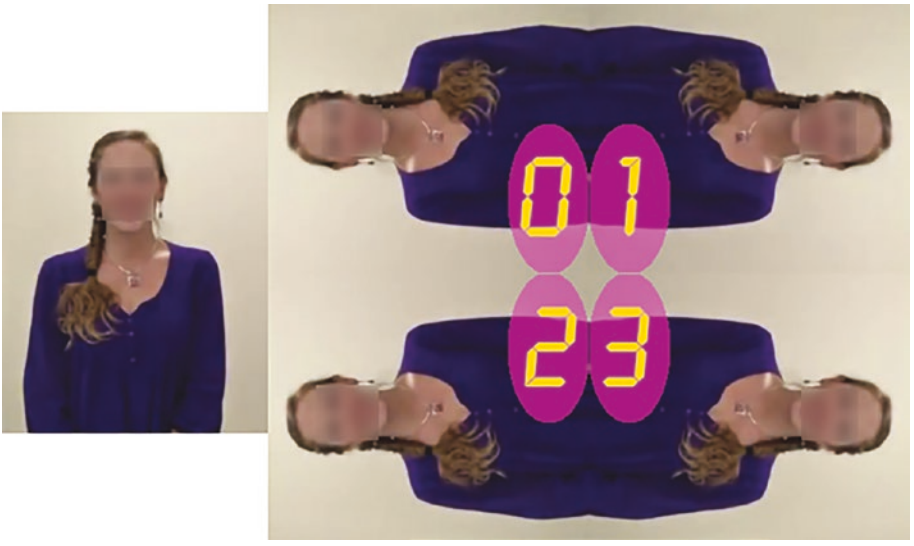


Figure 7-4. *These still images show* `dir` *values that can be used with the* `transpose` *filter*

The `transpose` filter option `passthrough` can have values `none`, `portrait`, and `landscape`. The value `none` is default. One of the last two values will be particularly useful in automated scripts to prevent unnecessary rotation, that is, when the video is already in the orientation specified by the `passthrough` filter option. It will also prevent `ffmpeg` from autorotating a video and then applying your transpose setting (causing further rotation).

You can rotate videos by more discrete levels than multiples of 90 degrees. The `rotate` filter accepts values in radians rather than degrees. The following `ffmpeg` command rotates a video by 16 degrees.

```
ffmpeg -y -i malampuzha-lake.mp4 \
       -filter_complex \
            "rotate=angle=16*PI/180:fillcolor=brown" \
       malampuzha-lake-tilt-16-chopped.mp4
# Rotates video but corners get cut off
```

☞ If the video becomes distorted, correct it using `setdar` filter.

☞ To convert degrees to radians, it has to be multiplied with $\pi/180$.

To prevent the corners from getting chopped off, the frame dimensions need to be increased. You can use the `rotw` and `roth` functions for determining these new dimensions. The two functions use these formulas internally.

$$\text{rotw}(\theta) = \text{Height} \times \text{Sine}(\theta) + \text{Width} \times \text{Cosine}(\theta)$$
$$\text{roth}(\theta) = \text{Width} \times \text{Sine}(\theta) + \text{Height} \times \text{Cosine}(\theta)$$

```
# Rotate video and enlarge the frame to prevent
# corners from getting cut off
ffmpeg -y -i malampuzha-lake.mp4 \
       -filter_complex \
          "rotate=angle=16*PI/180:
           ow=trunc(rotw(16*PI/180)/2)*2:
           oh=trunc(roth(16*PI/180)/2)*2:
           fillcolor=brown" \
       malampuzha-lake-tilt-16.mp4
```

As FFmpeg requires that the new width and height be even numbers, that is, divisible by 2, the calculated dimensions are first divided by 2, truncated off, and then multiplied by 2.

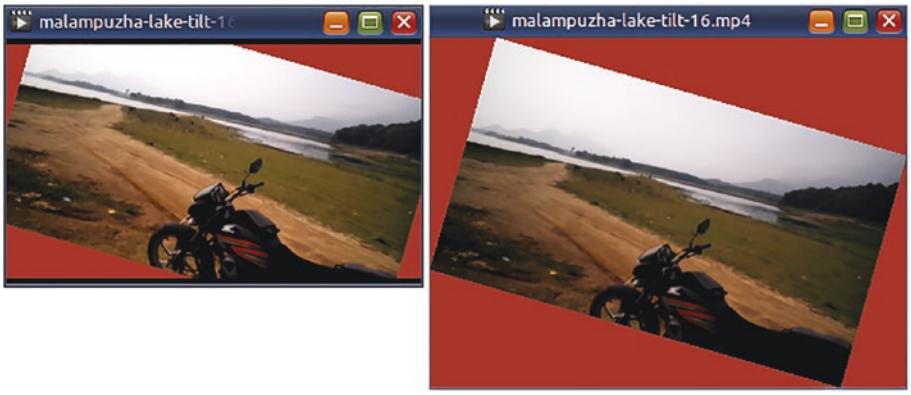


Figure 7-5. *The first video has the original dimensions, but the rotated content has chopped-off corners. The second video has bigger dimensions to accommodate the extruding corners*

Flip a Video

Some videos are flipped for some reason. Use `vflip` or `hflip` to set them right.

Figure 7-6. *These still images show which filter to us for what effect*

```
ffmpeg -i exhibit.mp4 \
       -filter:v "vflip" \
       exhibit-upside-down.mp4

ffmpeg -i exhibit.mp4 \
       -filter:v "hflip" \
       exhibit-half-crazy.mp4

ffmpeg -i exhibit.mp4 \
       -filter:v "hflip,vflip" \
       exhibit-totally-flipped.mp4
```

Brighten a Video (Adjust Contrast)

It is inevitable that some of your videos are dark, even when they were captured in broad daylight. You can use the `eq` filter to adjust the brightness. However, adjusting the brightness requires a subsequent adjustment of the contrast. The ranges for the options of this filter are listed in Table 7-2.

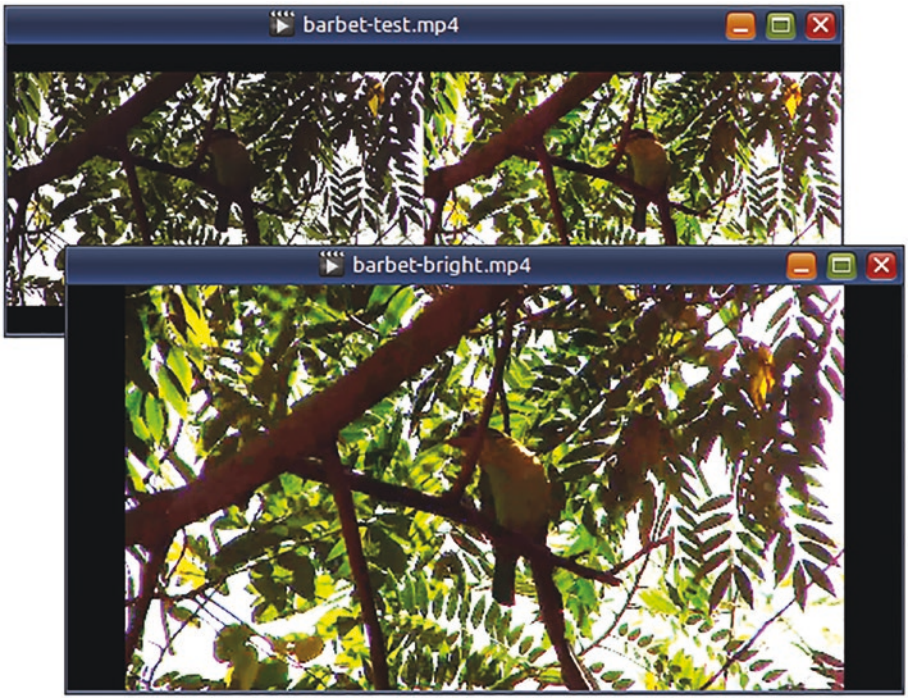


Figure 7-7. *After cumulative applications of brightness, saturation, and contrast filters, more detail of the green barbet is visible. Forget the background*

First, I decided to do a side-by-side comparison.

```
ffmpeg -y -i barbet.mp4 \
  -filter_complex \
    "[0:v]pad=(iw*2):ih:0:0[frame];
     [0:v]eq=brightness=0.2[bright];
     [bright]eq=saturation=3[color];
     [color]eq=contrast=2[dark];
     [frame][dark]overlay=W/2:0[out]" \
  -map '[out]' -map 0:a \
  barbet-test.mp4
```

Table 7-2. *Options for filter eq*

Filter option	Lowest	Highest	Default
Brightness	-1	1.0	0
Contrast	-1000	1000	1
Saturation	0	3	1
Gamma	0.1	10	1

After some trial-and-error attempts, I applied the filters to the original video.

```
ffmpeg -y -i barbet.mp4 \
  -filter_complex \
    "[0:v]eq=brightness=0.2[bright];
     [bright]eq=saturation=3[color];
     [color]eq=contrast=2[dark]" \
  -map '[dark]' -map 0:a \
  barbet-bright.mp4
```

Generate a Test Video

In the good old days, when there was just one TV channel in India, the transmission began in the evening with a 30-minute video test – something like this!

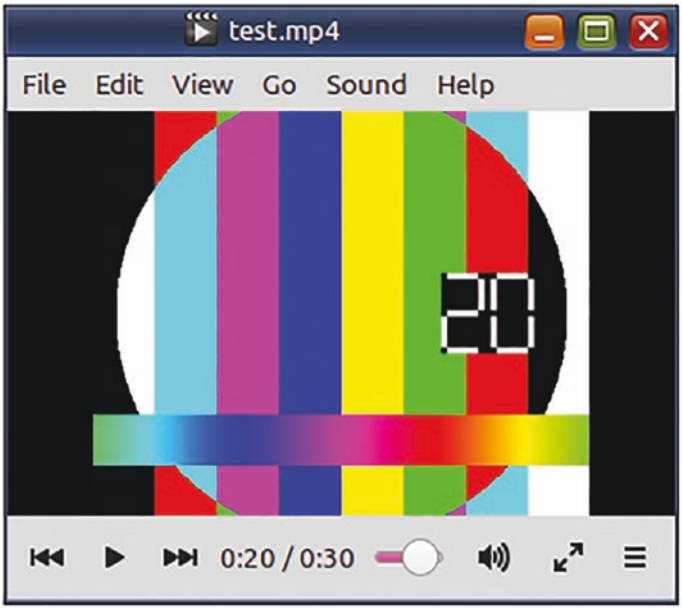


Figure 7-8. *The* `testsrc` *filter is a source filter that generates a test video stream*

The test video has a color pattern, a scrolling gradient, and a changing timestamp. The audio is a low white noise. I do not know who needs this video, but if it floats your boat, then here is the command to create it.

```
ffmpeg -f lavfi \
       -i "testsrc=size=320x260[out0];
           anoisesrc=amplitude=0.06:color=white[out1]" \
       -t 0:0:30 -pix_fmt yuv420p \
       test.mp4
```

☞ This command uses a set of filters as a pseudo file source (`-f lavfi`). It requires that the filter outputs be labeled `out0, out1, out2,...`.

☞ Filters whose name end in "`src`" are *source filters*. They do not require an input stream.

Remove Logo

In 2019, a newspaper in New York published an opinion alleging bias against women in government experiments. NASA's Apollo Space Program was then celebrating its 50th anniversary.

```
ffmpeg -i apollo-program.mp4 \
       -filter:v "delogo=x=520:y=10:w=100:h=50" \
       apollo-program-you-are-dead.mp4
```

Figure 7-9. *With the* `delogo` *filter, it is very easy to remove an unwanted logo from a video*

☞ After applying the filter, the logo has disappeared from the top-right corner.

☞ This video is only a simulation.

Fade into Another Video (And in Audio Too)

In order to prove aliens do not exist and have fun while doing that, I took videos from two authoritative US government agencies – NASA and IRS. The videos are in public domain, as the agencies are taxpayer-funded. The NASA video clearly states that there are no aliens, but I am not interested in their explanation. The IRS video is a tax advisory for noncitizens, also known as aliens. That is the fun part. In the output video, the first video plays fine until six seconds after which it fades out in three seconds. As the first video fades away, the second video starts fading in for three seconds. After that, it plays for six seconds.

Figure 7-10. *These screenshots show the crossfade sequence involving the two input videos*

Mixing these two videos can be done with one command, but for clarity, I have split it into four commands. (You should combine the filters to avoid multiple re-encoding.) The crossfade effect is performed by the `fade` filter for video and the `afade` filter for audio. The `trim` and `atrim` filters are used to divide the video and audio tracks into two parts – one where the stream plays normally and another where the fade filters take effect. I used `overlay` and `amix` filters to mix the second parts. After that, the `concat` filter was used to put three segments together – normal playback from the first file, crossfade effect from both files, and then normal playback from the second file.

```
# Make the second video same size as the first
ffmpeg -y -i irs-tax-advice-for-alien-mates.mp4 \
 -filter:v "pad=w=640:h=ih:x=(ow-iw)/2:y=0:color=yellow,
            fps=24" \
 -t 0:0:20 -pix_fmt yuv420p \
 irs-tax-advice-for-alien-mates2.mp4

# Create the fade-in-fade-out video
ffmpeg -y -i Do-Aliens-Exist-We-Asked-a-NASA-Scientist.mp4 \
 -i irs-tax-advice-for-alien-mates2.mp4 \
 -filter_complex \
  "[0:v:0]trim=start=0:end=6, setpts=PTS-STARTPTS, fps=24[v1];
   [1:v:0]trim=start=3:end=9, setpts=PTS-STARTPTS, fps=24[v2];
   [0:v:0]trim=start=6:end=9, setpts=PTS-STARTPTS, fps=24[v3];
   [1:v:0]trim=start=0:end=3, setpts=PTS-STARTPTS, fps=24[v4];
   [v3]fade=t=out:d=3:alpha=1, setpts=PTS-STARTPTS,
     fps=24[nasafade];
   [v4]fade=t=in:d=3:alpha=1, setpts=PTS-STARTPTS,
     fps=24[irsfade];
   [nasafade][irsfade]overlay, setpts=PTS-STARTPTS,
     fps=24[fading];
```

```
   [v1][fading][v2]concat=n=3:v=1:a=0[v]" \
 -map '[v]' -pix_fmt yuv420p \
 aliens-r-us-v.mp4

# Create the fade-in-fade-out audio
ffmpeg -y -i Do-Aliens-Exist-We-Asked-a-NASA-Scientist.mp4 \
    -i irs-tax-advice-for-alien-mates2.mp4 \
    -vn \
    -filter_complex \
     "[0:a:0]atrim=start=0:end=9, asetpts=PTS-STARTPTS[a1];
      [1:a:0]atrim=start=0:end=9, asetpts=PTS-STARTPTS[a2];
      [a1][a2]acrossfade=duration=3" \
     aliens-r-us-a.m4a

# Mix the video and audio
ffmpeg -i aliens-r-us-v.mp4 -i aliens-r-us-a.m4a \
       -codec copy \
       aliens-r-us.mp4
```

Crop a Video

For some screenshots in the beginning of this chapter, I needed a public-domain video of a sign-language translator. I found one but it was too big. I grabbed a still image from the video using a media player and edited it in GIMP.

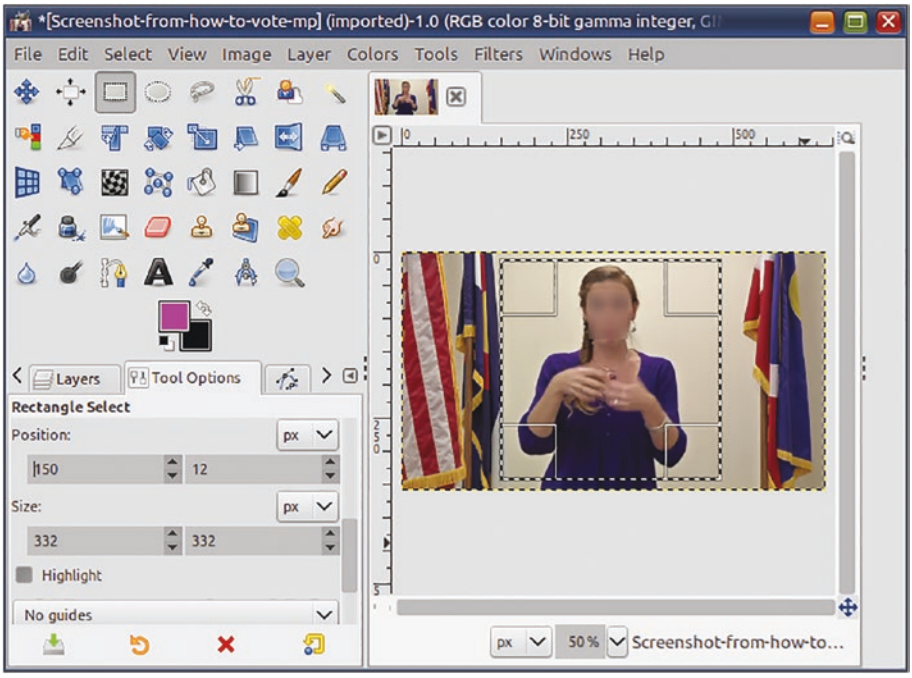


Figure 7-11. *First, take a screengrab from the video. Then, use an image-editing program to identify the location (150,12) and dimensions (332,332) of the region you want to cut out*

I then selected the region that I wanted cut into. I noted down the coordinates and dimensions of the region from GIMP's *Tool Options* panel. I used the details from GIMP in the options for a `crop` filter that I used on the video.

```
ffmpeg -i how-to-vote.mp4 \
       -filter:v "crop=332:332:150:12" \
       accessibility.mp4
```

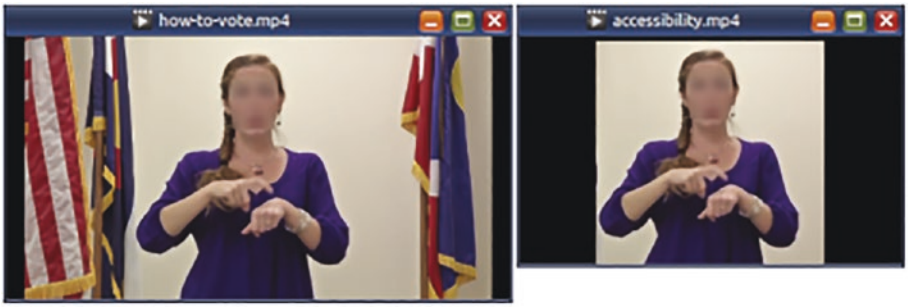


Figure 7-12. *The* `crop` *filter cut into a portion of a video*

Blur or Sharpen a Video

When this video was shot, there was a lot of camera refocusing and the action was blurry. The `smartblur` filter almost fixes this when it is set to *sharpen* the video.

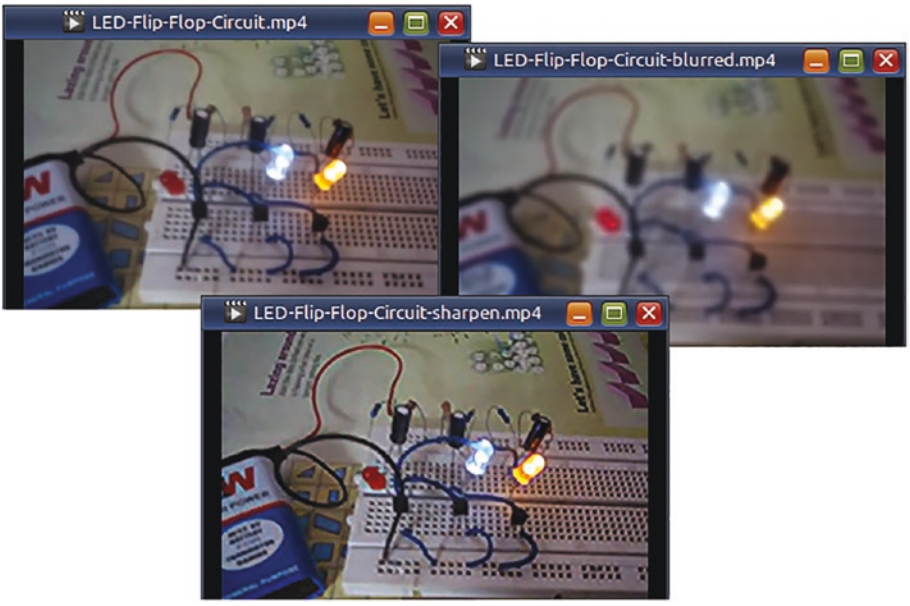


Figure 7-13. *With the* `smartblur` *filter, you can blur or sharpen a video*

```
ffmpeg -i LED-Flip-Flop-Circuit.mp4 \
       -filter:v
         "smartblur=luma_radius=5:luma_strength=1.0:
          luma_threshold=30" \
       LED-Flip-Flop-Circuit-blurred.mp4

ffmpeg -i LED-Flip-Flop-Circuit.mp4 \
       -filter:v
         "smartblur=luma_radius=5.0:luma_strength=-1.0:
         luma_threshold=30" \
       LED-Flip-Flop-Circuit-sharpen.mp4
```

The `smartblur` filter can blur or sharpen videos *without affecting the outlines*. It works on the brightness of the pixels. The `luma_radius` (0.1 to 5) represents the variance of the Gaussian blur filter. `luma_strength` (-1 to 1) varies between sharpness to blurring. `luma_threshold` (-30 to 30) varies the focus of the filter from the edges to interior flatter areas.

Blur a Portion of a Video

Sometimes, you need to protect the identity of some people (e.g., bystanders) who are not really the focus of a video. Use the `boxblur` filter. This command tries to blur two regions in a video where human faces appear.

```
ffmpeg -y -i stilt.mp4 \
  -filter_complex \
    "[0:v]crop=260:80:400:550[c1];
     [0:v]crop=100:60:1:550[c2];
     [c1]boxblur=6:6[b1];
     [c2]boxblur=6:6[b2];
     [0:v][b1]overlay=400:550[v1];
```

```
    [v1][b2]overlay=1:550[v]" \
  -map '[v]' -map 0:a -c:a copy \
 stilt-masked.mp4
```

Unlike `smartblur`, it does not respect object outlines. And, contrary to its name, `boxblur` does not blur inside the box or a part of the video. It affects the whole frame of the input video stream.

Figure 7-14. *With the* `boxblur` *filter, you can blur content without discrimination of any outlines*

☞ To avoid any doubt or confusion, I would like to state that I have masked faces of private individuals (even in public-domain content) in several screenshots using an image-editing program. In this screenshot, however, the effect was achieved using the `ffmpeg` filter `boxblur`.

Draw Text

To draw text on video, you need to use the `drawtext` filter and also specify the location of the font file. When you are drawing several pieces of text, it is better to daisy-chain your texts (using commas, not semicolons).

```
ffmpeg -y -i color-test.mp4 \
  -filter_complex \
    "[0:v:0]drawtext=x=(w-tw)/2:y=10:fontcolor=white: \
        shadowx=1:shadowy=1:text='Detonation Sequence': \
        fontsize=25: fontfile=AllertaStencil.ttf, \
     drawtext=x=(w-tw)/2:y=60:fontcolor=white: \
        shadowx=1:shadowy=1: \
        text='This TV will self-destruct in t seconds.': \
        fontsize=15:fontfile=Exo-Black.ttf[v]" \
  -map '[v]' -map 0:a:0 -pix_fmt yuv420p \
  idiot-box-1.mp4
```

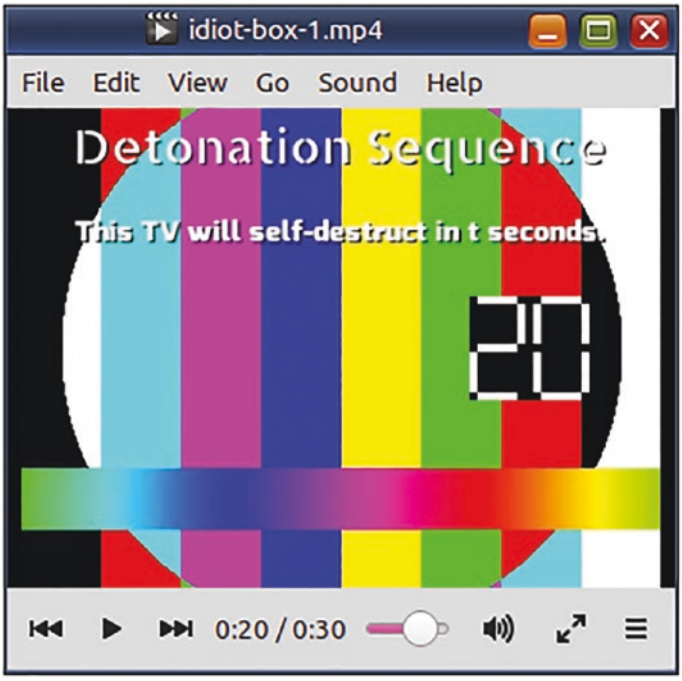


Figure 7-15. *With the* `drawtext` *filter, you can draw text formatted with fonts, styles, shadows, transparencies, etc. on video*

Draw a Box

You can use the `drawbox` filter to render all kinds of boxes, filled or bound, with all sorts of colors and transparencies.

```
ffmpeg -y -i color-test.mp4 \
 -filter_complex \
   "[0:v:0]drawbox=x=20:y=3:w=280:h=36:color=tomato@0.4:
   t=fill, \
    drawbox=x=11:y=49:w=294:h=40:color=lime:t=1, \
    drawtext=x=(w-tw)/2:y=10:fontcolor=white: \
        shadowx=1:shadowy=1:text='Detonation Sequence': \
```

```
        fontsize=25: fontfile=AllertaStencil.ttf, \
    drawtext=x=(w-tw)/2:y=60:fontcolor=white: \
        shadowx=1:shadowy=1: \
        text='This TV will self-destruct in t seconds.': \
        fontsize=15:fontfile=Exo-Black.ttf[v]" \
-map '[v]' -map 0:a:0 -pix_fmt yuv420p \
idiot-box-2.mp4
```

The part of the color value after the `@` symbol refers to the transparency level. It ranges from 0 (fully transparent) to 1 (opaque). If you specify the value `fill` for the filter option `t` or `thickness`, then the box will be filled with that color. Otherwise, it applies to the border.

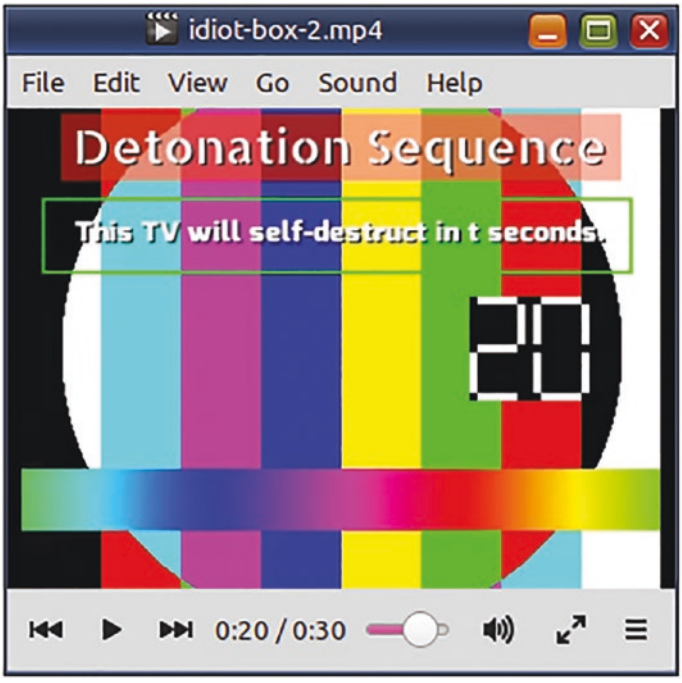


Figure 7-16. *With the* `drawbox` *filter, two rectangles around the text. (See original video in previous section.) The first rectangle is filled with red. The second rectangle is bordered green*

Speed Up a Video

When you increase the playback speed of a video, its duration decreases. When you slow down a video, its duration increases. There is no one filter that changes the speed of both the audio and the video. You need to use two different filters – one for video and one for audio. The two filters do not work in the same way. The two need to be calibrated correctly so that the same effect is achieved on both the audio and the video.

For the video, you need to set the `setpts` video filter to a fraction of the `PTS` filter constant. If you want to double the speed of the video, divide PTS by 2. If you want the video to be four times fast, then divide PTS by 4. For the audio, you need to use the `atempo` filter. The range of this filter is from half the speed to 100 times. The following command fast-forwards a video by four times (4x).

```
ffmpeg -y -i barb.mp4 \
  -filter_complex \
    "[0:v]setpts=PTS/4[v];
     [0:a]atempo=4[a]" \
  -map '[v]' -map '[a]' \
  barb-speed.mp4
```

☞ In older versions of FFmpeg, the maximum limit of the `atempo` filter was just 2. To go beyond that limit, multiple filters had to be daisy-chained: `atempo=2, atempo=2`

Slow Down a Video

In the *Tom & Jerry* film *Baby Puss*, one of the alley cats tries to dance with a seemingly innocuous doll. In the middle of it, I thought, the doll had become possessed and slammed the cat down on the floor! I slowed the video down with Ffmpeg, and my suspicions were confirmed.

To slow down a video, you need to use the same filters as in the previous section, but the multipliers will have to be different.

This command slows down the video and the audio to one-fourth.

```
ffmpeg -y -i tom.mp4 \
  -filter_complex \
    "[0:v]setpts=PTS*4[v];
     [0:a]atempo=0.5, atempo=0.5[a]" \
  -map '[v]' -map '[a]' \
  possessed-doll.mp4
```

☞ Note the different multiples used for video and audio to achieve the same effect. The audio filter has been used twice because of the limitation in its range.

☞ Read previous section for more information on these two filters.

Laurie Lennon, from the Lennon Sisters family, has published a tribute video for the *Merrie Melodies* number "Oh, Wolfie!". When I saw it for the first time some years ago, I felt the tempo was too high. I slowed the audio down in Audacity. (I have all songs featuring *Lou* as MP3 files, complete with Wolfie's and Droopy's crazy antics.) For my 2020 book, I tried to do

the same using FFmpeg and apply the change to the video as well. My calculation became easier when I used seconds. The original video was 114 seconds, and my slowed-down audio was 128 seconds.

```
# 128/114 and 114/128
ffmpeg -y -i Laurie-Lennon-Original.mp4 \
    -filter_complex \
    "[0:v]setpts=PTS*(128/114)[v];
     [0:a]atempo=(114/128)[a]" \
  -map '[v]' -map '[a]' \
  Laurie-Lennon-Slow.mp4
```

The links to these videos and those used in other examples in this book are available online:

```
www.vsubhash.in/ffmpeg-book.html
```

Summary

The examples in this chapter would have amply demonstrated that a lot of useful and powerful multimedia-processing abilities are hidden in the filters functionality. You need to read the relevant documentation to make full use of a filter. *Filter expressions* using built-in real-time variables (*filter constants*) and functions provide a lot of versatility and extensibility to command-line users that would have otherwise been limited to programmers who use the `libav` *libraries*.

In this book, the teaching portion about FFmpeg functionality ends here. The subsequent chapters are topic-specific for those who want quick answers to a particular type of problem and do not want to read through dense explanatory text before finding the answer. You will find some information repeated or not mentioned at all.

CHAPTER 8

All About Audio

In this chapter, you will learn to perform several tasks related to audio content. While it is convenient to have a separate chapter just for audio, you will find some information repeated from other chapters. If there is no explanation, then it must be self-explanatory.

Most audio-related tasks can be performed using audio filters. If any of the filters used in this chapter seem too complicated, find out what the official FFmpeg documentation has to say on them. If you are unfamiliar with using filters, read Chapter 7.

Convert from One Audio Format to Another

```
ffmpeg -i alarm.ogg \
       -c:a libmp3lame \
       -ac 2 \
       -b:a 128K \
       alarm.mp3            # Ogg to MP3
```

Extract Audio from a Video

```
ffmpeg -i music-video.mp4 \
       -c:a libmp3lame \
       -ac 2 \
       -b:a 128K \
       music-video.mp3      # Audio saved as MP3
```


V. Subhash, *Quick Start Guide to FFmpeg*, https://doi.org/10.1007/978-1-4842-8701-9_8

Convert a MIDI File to MP3 or Ogg

You may have noted that there are no codecs for MIDI. That is because MIDI files are quite different from ordinary sound files. Ordinary sound files contain the wave form encoded in a predefined format. In contrast, MIDI files are merely a collection of references to a common sound bank.

Timidity is the Linux way of playing MIDI files. You can use Timidity to playback MIDI files in WAVE format and write it to its *standard output*. Simultaneously, FFmpeg can be made to consume the wave output as its input file (from its *standard input* over a pipe) and convert it as a regular sound file.

```
timidity yamaha.midi -Ow -o - | ffmpeg -i - -b:a 128k
yamaha.ogg
```

The `-Ow` makes Timidity to output the playback in WAVE format. Its `-o` option is used to specify the output file. Instead of an output file, we use `-` to make it write to the *standard output*. The Timidity output is then piped over to an FFmpeg command, where it is captured from the *standard input* with yet another `-` (hyphen).

Change Volume

FFmpeg can increase the loudness of an audio file using its `volume` filter. The filter accepts a multiple either as a number (scalar) or in decibels (logarithmic).

```
ffmpeg -i sarah.mp3 -af 'volume=3' sarah-more.mp3
```

I had an audio file that continued to have low volume, even after trebling the levels. I opened it in Audacity and found the reason.

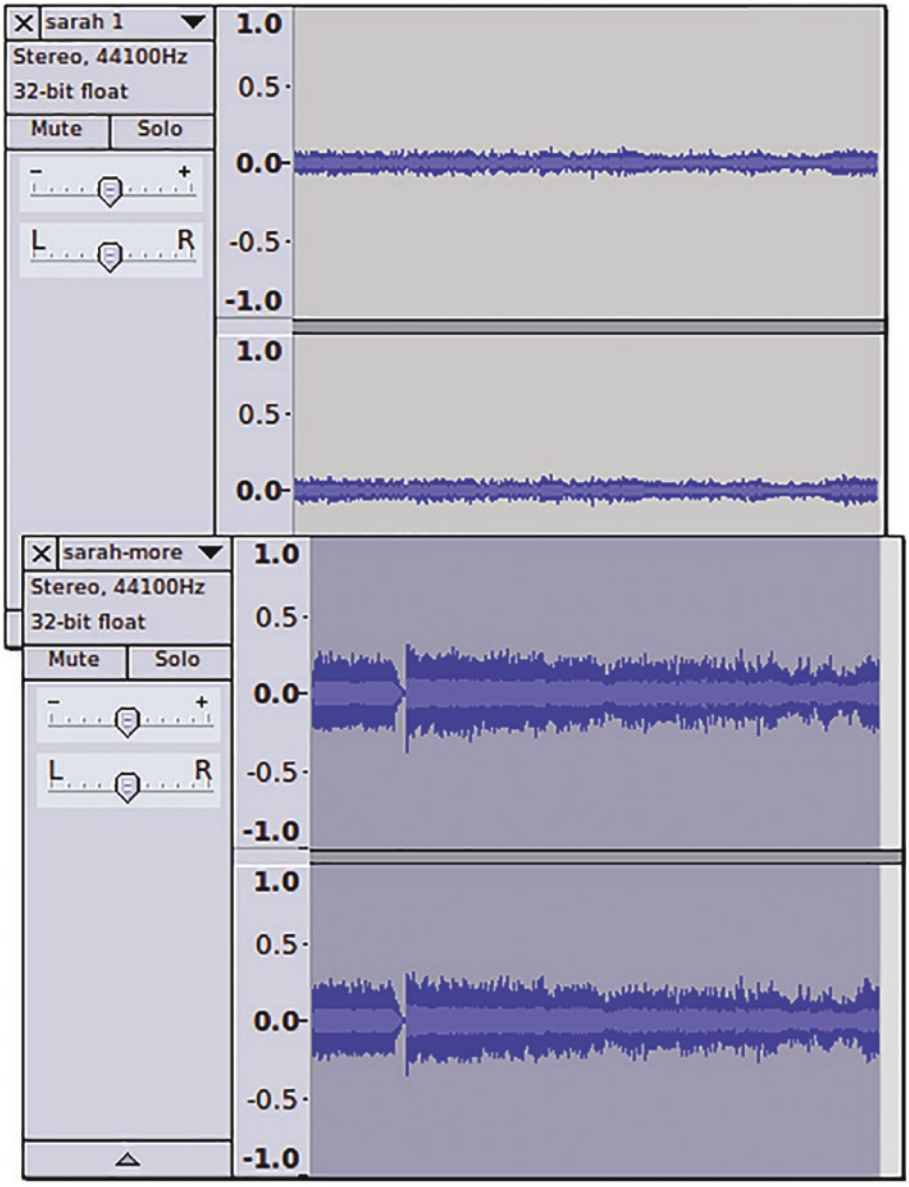


Figure 8-1. *Audacity confirms that irrationally increasing the volume is not making much of a difference*

Increasing sound like this is based on guesswork. It might work. It may also damage your hearing and/or your speaker system. The correct approach is to normalize the sound after observing the decibel levels in the current waveform.

```
ffmpeg -i sarah.mp3 -af "volumedetect" -f null -
```

```
~/Desktop
$ ffmpeg -i sarah.mp3 -af "volumedetect" -f null /dev/null
[Parsed_volumedetect_0 @ 0x226c100] mean_volume: -32.4 dB
[Parsed_volumedetect_0 @ 0x226c100] max_volume: -17.3 dB
[Parsed_volumedetect_0 @ 0x226c100] histogram_17db: 6
[Parsed_volumedetect_0 @ 0x226c100] histogram_18db: 15
[Parsed_volumedetect_0 @ 0x226c100] histogram_19db: 56
[Parsed_volumedetect_0 @ 0x226c100] histogram_20db: 452
[Parsed_volumedetect_0 @ 0x226c100] histogram_21db: 1676
```

Figure 8-2. *Run the* `volumedetect` *filter before increasing the volume. It helps you in determining the highest number of decibels to which the volume can be increased without cutting into the waveform*

☞ The `volumedetect` filter outputs text data to the *standard output*. It does not create an audio stream.

The `volumedetect` filter shows that we can safely increase the volume to 16db. If we raised the volume to 17dB or higher, normalization would cut into the waveform, and the peaks would get attenuated or chopped off. At 17dB, six sound samples (the loudest) in the waveform would be lost.

```
ffmpeg -i sarah.mp3 \
       -af 'volume=16dB' -f ogg \
       sarah-normalized.ogg
```

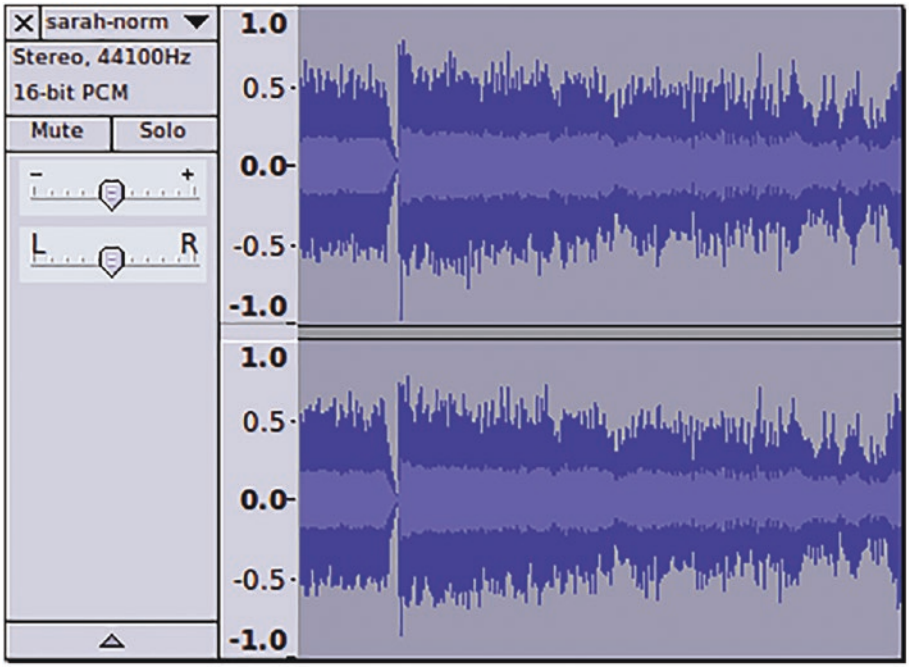


Figure 8-3. *Audacity confirms that the volume has been increased without cutting into the waveform*

This is fine. Now, how do you decrease the volume? Well, choose a fraction between 0 and 1 for the `volume` filter. For example, to decrease the volume by two-thirds, you should set the multiple at 0.33. (You know ⅓ = 0.33?)

```
ffmpeg -i sarah-normalized.ogg -af 'volume=0.33' sarah-less.mp3
```

Change Volume in a Video File

Say, to irrationally increase the volume by three times,

```
ffmpeg -i sarah.mp4 \
       -c:v copy \
       -af 'volume=3' \
       -c:a libmp3lame -b:a 128k \
       sarah-more.mp4
```

To safely and intelligently increase the volume in a video file,

```
ffmpeg -i sarah.mp4 \
       -af 'volumedetect' \
       -vn \
       -f null \
       /dev/null
# Displays that the loudest samples are at 17dB

# Increase the volume to 16dB (to safely normalize the audio)
ffmpeg -i sarah.mp4 \
       -c:v copy \
       -af 'volume=16dB' \
       -c:a libmp3lame -b:a 128k \
       sarah-normalized.mp4
```

To decrease volume by two-thirds in a video file, you need to use fractions:

```
# Reduces volume by two-thirds (or to one-thirds)
ffmpeg -i sarah-normalized.mp4 \
         -c:v copy \
         -af 'volume=0.33' \
         -c:a libmp3lame -b:a 128k \
         sarah-less.mp4
```

Dynamic Range Compression/Normalization

Sometimes, normalization does not make any difference. The volume seems to be unchanged. Examining the audio in Audacity can show you the problem. There are volume spikes in some locations while much of the file is at low volume. (These spikes usually occur when the mic is shaken or bumped while it is recording.) Normalization cannot proceed as long as the spikes remain. The solution is to identify the low-volume regions and expand the waveform. This more selective normalization is known as Dynamic Range Normalization. Alternatively, you could bring down the high-volume regions to the level of the rest of the audio. This selective compression of the waveform is known as Dynamic Range Compression (DRC).

Both techniques make irreversible changes to the waveform, so do not use them indiscriminately. DRC is the bane of popular music today and makes it very boring.

In Carl Orff's composition of *O Fortuna* or Ryuichi Sakamoto's score for the end credits of the movie *Femme Fatale*, the music starts on a low note, building slowly in a steady crescendo and abruptly drops off a high cliff. Applying DRC on such an audio would ruin the composer's intent. However, a recording of a teleconferencing session where multiple participants are heard speaking at different volumes would be an ideal candidate for DRC.

The `dynaudnorm` filter can perform both functions, but the default is normalization. When the `guasssize` option is set at the lower end of 3, it behaves like a typical compressor. At the other end of 300, it becomes a traditional normalizer.

```
ffmpeg -y -i train-trip-low.mp3 \
       -filter:a dynaudnorm=gausssize=3 \
       train-trip-low-dynaudnormalized.mp3
```

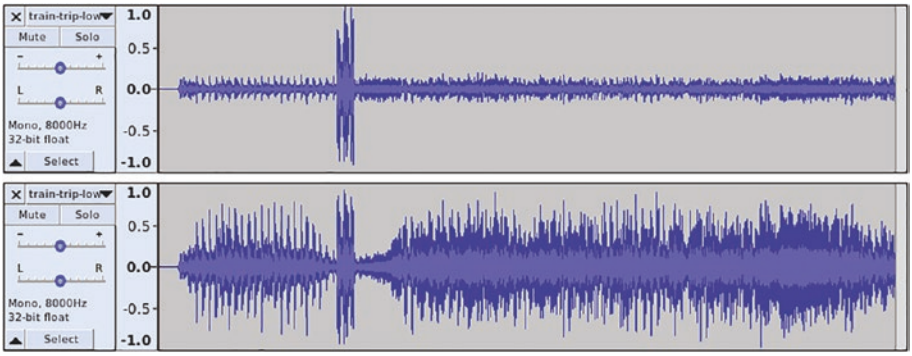


Figure 8-4. *A few unexplained spikes in volume can prevent normalization from happening on the rest of the waveform. Dynamic Range Compression and Dynamic Range Normalization are not affected by these spikes and change the entire waveform*

Channels

An audio stream can have one or more channels. A *channel* is an independent sequence of audio. All channels in an audio stream are of the same length, and they are played back simultaneously. The idea of having a separate channel is to have a different choice of musical instruments or sounds to play in different speakers. Audio content creators may move back and forth sounds between different channels at different volume levels. This can be useful in creating a 2D or 3D effect to the sound. Typically, each channel in an audio stream is assigned to a particular speaker. This composition of channels in a multichannel stream is known as its *channel layout*. When the number of speakers is less than the number of channels, then that particular channel may not be heard, or the device may *downmix* the channels so that the excess channels will be heard on the existing speakers.

Monaural audio has only one channel. Stereo music has two channels – left and right. Movies can have two, six, seven, eight, or more channels.

When working with channels, you will need to use filters such as `amerge`, `channelmap`, `channelsplit`, and `pan`. These filters make use of certain IDs for channels and channel layouts. Table 8-1 and Table 8-2 list these IDs.

Table 8-1. *Channels*

ID	Channel
FL	Front left
FR	Front right
FC	Front center
LFE	Low frequency
BL	Back left
BR	Back right
FLC	Front left-of-center
FRC	Front right-of-center
BC	Back center
SL	Side left
SR	Side right
TC	Top center
TFL	Top front left
TFC	Top front center
TFR	Top front right
TBL	Top back left
TBC	Top back center
TBR	Top back right
DL	Downmix left
DR	Downmix right
WL	Wide left
WR	Wide right
SDL	Surround direct left
SDR	Surround direct right
LFE2	Low frequency 2

Table 8-2. *Channel layouts*

ID	Layout composition
Mono	FC
Stereo	FL+FR
2.1	FL+FR+LFE
3.0	FL+FR+FC
3.0(back)	FL+FR+BC
4.0	FL+FR+FC+BC
Quad	FL+FR+BL+BR
Quad(side)	FL+FR+SL+SR
3.1	FL+FR+FC+LFE
5.0	FL+FR+FC+BL+BR
5.0(side)	FL+FR+FC+SL+SR
4.1	FL+FR+FC+LFE+BC
5.1	FL+FR+FC+LFE+BL+BR
5.1(side)	FL+FR+FC+LFE+SL+SR
6.0	FL+FR+FC+BC+SL+SR
6.0(front)	FL+FR+FLC+FRC+SL+SR
Hexagonal	FL+FR+FC+BL+BR+BC
6.1	FL+FR+FC+LFE+BC+SL+SR
6.1	FL+FR+FC+LFE+BL+BR+BC
6.1(front)	FL+FR+LFE+FLC+FRC+SL+SR
7.0	FL+FR+FC+BL+BR+SL+SR
7.0(front)	FL+FR+FC+FLC+FRC+SL+SR
7.1	FL+FR+FC+LFE+BL+BR+SL+SR
7.1(wide)	FL+FR+FC+LFE+BL+BR+FLC+FRC
7.1(wide-side)	FL+FR+FC+LFE+FLC+FRC+SL+SR
Octagonal	FL+FR+FC+BL+BR+BC+SL+SR
Hexadecagonal	FL+FR+FC+BL+BR+BC+SL+SR+WL+WR+TBL+TBR+TBC+TFC+TFL+TFR
Downmix	DL+DR
22.2	FL+FR+FC+LFE+BL+BR+FLC+FRC+BC+SL+SR+TC+TFL+TFC+TFR+TBL+TBC+TBR+LFE2+TSL+TSR+BFC+BFL+BFR

Swap Left and Right Channels

In some videos, sounds from the left side of the video are heard on the right channel and those from the right side are on the left channel. In such a case, you can do a switcheroo.

```
# Switch right and left channels of stereo audio
ffmpeg -i wrong-channels.mp4 \
       -c:v copy \
       -filter_complex "channelmap=map=FR-FL|FL-FR" \
       fine-channels.mp4
```

You can specify the channel settings using the `map` filter option in this format:

```
       input_channel_id-output_channel_id|input_channel↵
       _id-output_channel_id|...
```

This filter also has a `channel_layout` option.

Turn Off a Channel

In some video files, the narration or commentary is on one channel, and the ambient noise or background music is on the other. If what you want is on the left, you can turn the right channel off by setting its gain to zero (0).

```
# Silence right channel
ffmpeg -i moosic.mp3  \
       -c:v copy \
       -filter_complex "pan=stereo|FL=FL|FR=0" \
       moosic4lefty.mp3
```

☞ Changing the audio to mono (single-channel audio) is not an option because mono audio is played on both front and left speakers.

You can specify the channel settings in this format:

```
l|output_channel_id=gain*input_channel_id|output⏎
_channel_id=gain*input_channel_id...
```

The filter option `l` is used to specify the channel layout. After that, you have to specify how much of what channel (in the input stream) you need for each channel in the output audio stream. For specifying that proportion or the gain, you can specify a multiple or a fraction. If you omit the gain, it implies that you want that channel as is or that the gain is equal to 1 (one). If you use 0 (zero), it means that you want that channel totally attenuated.

Move Channel to a Separate Audio Track

In some videos, the left and right audio channels are independent tracks. What these content creators do is place the original audio on one channel and the most annoying royalty-free music on the other. Instead of deleting the offending channel, you could move each channel to a separate audio stream while preserving the original stereo stream in a third stream.

The `channesplit` filter has a `channel_layout` filter option which by default assumes the input audio stream is stereo. Because of that, this command splits the left and right channels of the audio stream in the video to two mono streams, which I have labeled as `L` and `R`.

```
# Split channels to separate audio streams
# and also preserve existing audio stream
ffmpeg -y -ss 0:0:20 -t 0:0:20 -i zombie.mp4 \
       -c:v copy \
       -filter_complex "channelsplit[L][R]" \
```

```
        -map 0:v:0  -map '[L]' -map '[R]' -map 0:a:0 \
        -c:a:0 aac -ac:a:0 1 \
        -c:a:1 aac -ac:a:1 1 \
        -c:a:2 copy \
        zombie-tracks.mp4
```

Because the first two of the mapped output audio streams need to be freshly encoded as mono streams and the last mapped audio stream just needs to be copied without re-encoding, encoder (`-c`) and channel count (`-ac`) need to be specified on a *per-stream* basis.

☞ The `-c` and `-ac` options are limited to the streams specified by the `-map` options specified before them.

Fix Out-of-Phase Audio Channels

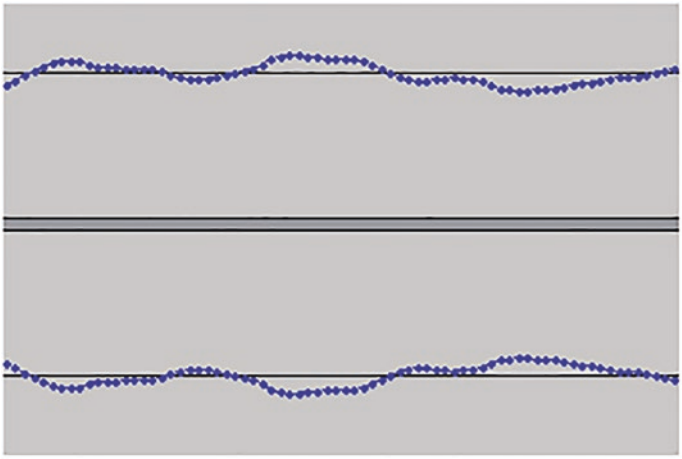


Figure 8-5. *This zoomed-in waveform shows out-of-phase left and right channels*

Rarely, when you downmix to mono sound, out-of-phase audio in the channels may cancel each other out. The audio will sound muted. You can fix it by saving either the left or the right channel in the input file as the only (mono) channel in the output file. (Monaural audio is played the same on both sides.)

Change Stereo to Mono

Stereo audio has two channels – left and right. Most of the time, both channels have the same audio. However, in many cases, the left channel will have some sounds that are not available in the right channel. The loudness of certain sounds may also differ. This difference will be lost when you convert to mono. Remember this before converting to mono. Mono audio cannot be converted back to stereo. It can only be made to look like stereo. You can convert stereo to mono either by downmixing both left and right channels to a mono channel or dropping one of the channels.

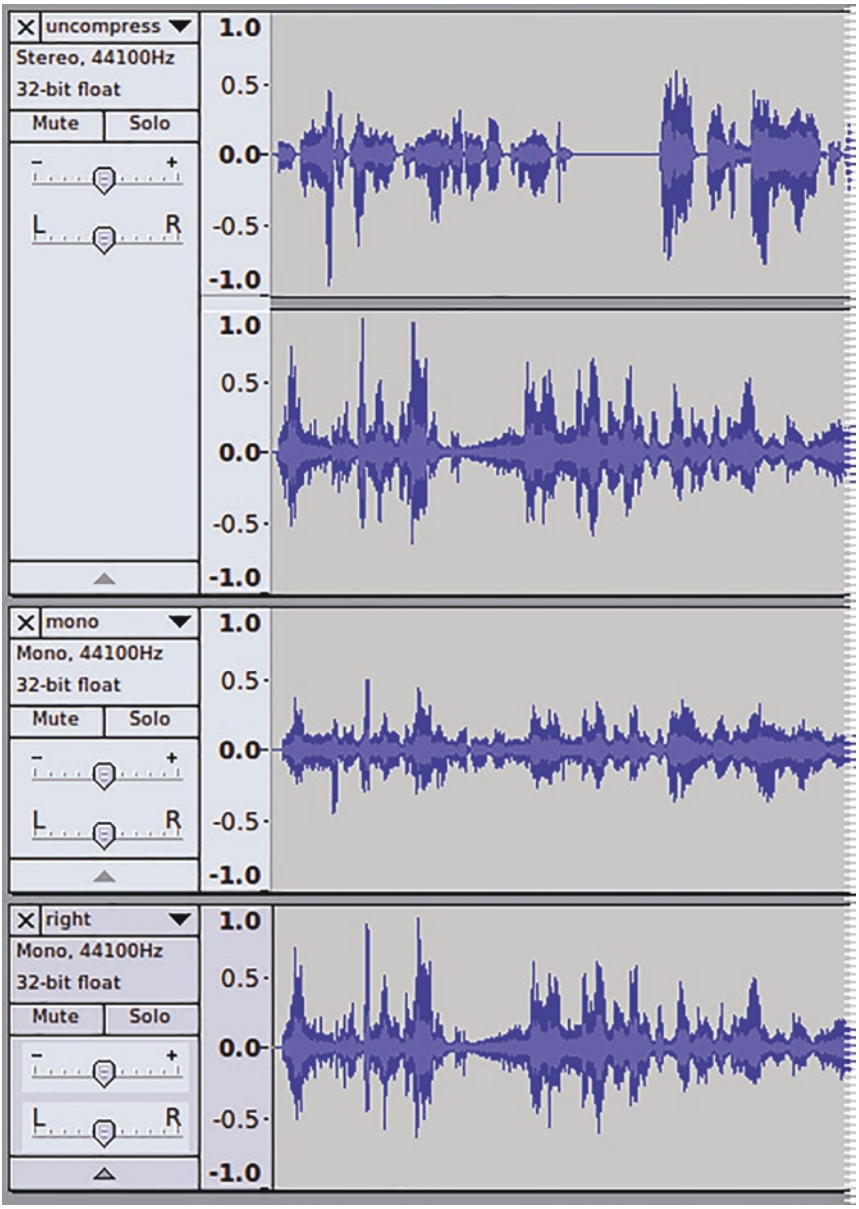


Figure 8-6. *To convert from stereo to mono, you can downmix left and right channels to a single mono channel or drop one of the channels. In either case, if the two channels are different, there will be some irreversible loss of the waveform*

```
# Downmix to mono
ffmpeg -i uncompressed-stereo.wav \
       -ac 1 \
       mono.mp3

# Drop left channel
ffmpeg -i uncompressed-stereo.wav \
       -filter channelmap=FR-FC:mono \
       right.mp3
```

Convert Mono to Stereo

Mono audio has only one channel. On a stereo audio output device, the same channel will anyway be played on the left and right speakers. Hence, it does not make any difference to convert mono to stereo. If at all this needs to be done, then the audio can be split with a second channel.

```
ffmpeg -i mono.mp3 \
       -ac 2 \
       stereo-kind-of.mp3
```

Make Audio Comfortable for Headphone Listening

When wearing headphones, the sounds feel like they are arising inside your head and between your ears. The `earwax` filter makes the sound feel like it is outside and in front of your head.

```
ffmpeg -i in-head.flac -filter "earwax" out-head.mp3

ffmpeg -i tl.mp4 -filter:a "earwax" -c:v copy tl-head.mp4
```

Downmix 5.1 Audio to Stereo

Using the `-ac` (audio channels) option with the necessary number of channels is enough for most downmixing operations.

```
ffmpeg -i AAC-LC-Channel-ID.mp4 \
       -ac 2 \
       stereo.mp3
```

Downmix Two Stereo Inputs to One Stereo Output

When you place two videos side-by-side each other, you need to do something about their two audio streams.

```
ffmpeg -y -i beto.mp4 -i fallon.mp4  \
  -filter_complex \
    "[0:v]pad=1280:360:0:0 [frame];
     [frame][1:v]overlay=640:0 [overlaid];
     [0:a]channelsplit=channel_layout=mono[beto];
     [1:a]channelsplit=channel_layout=mono[fallon];
     [beto][fallon]join=inputs=2:channel_layout=stereo[audio]" \
   -map '[overlaid]' -map '[audio]' \
   fallon-aces-beto.mp4

ffmpeg -y -i beto.mp4 -i fallon.mp4  \
  -filter_complex \
    "[0:v]pad=1280:360:0:0 [frame];
     [frame][1:v]overlay=640:0 [overlaid];
     [0:a][1:a]amerge=inputs=2[audio]" \
   -map '[overlaid]' -map '[audio]' \
   -ac 2 \
   fallon-aces-beto2.mp4
```

The first command uses `channelsplit` filter to convert stereo audio from the two input files to mono streams. It then uses the `join` filter to use the two mono streams to create a stereo stream where the mono audio from the first file is the left channel and the mono audio from the second file becomes the right channel.

The second command uses `amerge` filter to create a four-channel audio stream from the two input stereo (two-channel) streams. The `-ac 2` conversion setting downmixes the four-channel audio to a two-channel stereo output.

In the first command, the input audio streams are assumed to be of equal length. If they are not of equal length, then the `apad` filter needs to be used to add silence to last till the end of the video stream.

For the Laurie Lennon video mentioned in an earlier chapter, I had also created a video with both the original version and the slowed-down version side-by-side for comparison. The slowed-down video was of greater duration. Without adding the extra silence, FFmpeg would continue adding duplicate data at the end of the shorter stream. The process would never complete, and my computer would have run out of space.

```
// Slow MP4 was 128 seconds. The original was 114 seconds.
ffmpeg -i Laurie-Lennon-Slow.mp4 \
       -i Laurie-Lennon-Original.mp4 \
       -loop 1 -i bg.png \
    -filter_complex \
    "[0:v:0]scale=320:180[v1];
     [1:v:0]scale=320:180[v2];
     [2:v:0][v1]overlay=320:90[v3];
     [v3][v2]overlay=0:90[v];
     [0:a:0]channelsplit=channel_layout=mono[right];
     [1:a:0]channelsplit=channel_layout=mono,apad[left];
     [left][right]join=inputs=2:channel_layout=stereo[a]" \
```

```
  -map '[v]' -map '[a]'  \
  -t 0:2:08 \
  -y laurie-lennon-comparison.mp4
```

Render a Visual Waveform of the Audio

The `showwaves` filter renders a visual waveform of the input audio.

```
ffmpeg -y -i dialup-modem.mp4 \
  -filter_complex \
    "[0:a]showwaves=s=160x90:mode=line[waves];
     [0:v]drawbox=x=(iw-20-w):y=(ih-20-h):w=160:h=90:
            color=yellow@0.6:t=fill[bg];
     [bg][waves]overlay=x=(W-20-w):y=(H-20-h)[over]" \
  -map '[over]' -map 0:1 \
  dialup-modem-handshake.mp4
```

Figure 8-7. *This command draws a waveform of the dialup modem handshake tones on the video. To make the waveform easily visible, the command has drawn a translucent yellow box behind it*

In 2021, I wrote a book on electronics. In that, I described how to create the most annoying-sounding alarm noise using a blinking LED. I wanted to publish an online video of the alarm but felt queasy about posting a video of the ceiling where the alarm was installed. FFmpeg to the rescue! I used the `showfreqs` filter to generate the "power spectrum" of the audio recording.

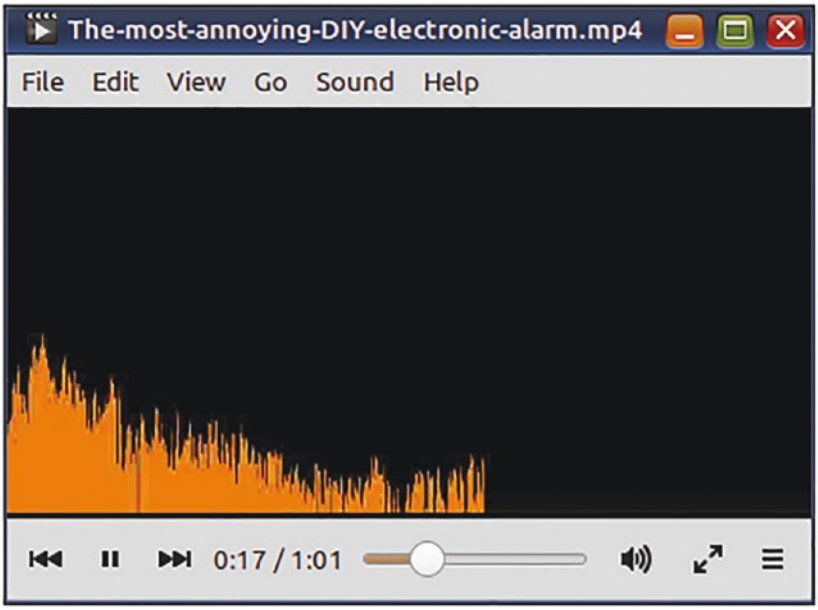


Figure 8-8. *The `showfreqs` filter shows how energy in an audio signal is spread across the range of frequencies that are audible to the human ear*

```
ffmpeg -i The-most-annoying-DIY-electronic-alarm.mp3 \
       -filter_complex \
         "showfreqs=s=640x320:mode=bar[v]" \
       -map '[v]' -map 0:a:0 \
       -c:v mpeg4 -b:v 466k -r 24 \
       The-most-annoying-DIY-electronic-alarm.mp4
```

There are a few other filters similar to this one. Check the documentation. These filters are very interesting.

Detect Silence

I have a shell script for censoring movies. (It uses FFmpeg, of course.) I use it to protect kids from foul dialog and unsuitable scenes. It asks for timestamps where the audio needs to be silenced and the video needs to be blacked out. After it does the job, I need to double-check these locations before the grand première on the TV. I use this command:

```
ffmpeg -i edited-movie.mp4 \
       -filter:a "silencedetect" \
       -vn -f null -
```

This command outputs timestamps wherever silence is detected. This helps me to directly skip to the censored locations using my media player on my computer.

Silence the Video

Heck, you do not want sound at all! Just remove the audio stream.

```
ffmpeg -i music-video.mp4 \
       -an \
       -c:v copy \
       sound-of-silence.mp4
```

Convert Text to Speech

If your `ffmpeg` executable has been built-in with support for the `libflite` text-to-speech synthesizer library, then you can convert text content to spoken words.

```
ffmpeg -f lavfi \
       -i "flite=textfile=speech.txt:voice=slt" \
       speech.mp3
```

This library has an option for a female voice, but I like the male-only `espeak` better. You can find other options for the `flite` filter option `voice` by typing the following:

```
ffprobe -f lavfi "flite=list_voices=1"
```

On my computer, this command lists `awb`, `kal`, `kal16`, `rms`, and `slt` as voices that are supported.

Apply a Low-Pass Filter

In an earlier chapter, I mentioned that I used Audacity to apply a *low-pass filter*. A low-pass filter makes all frequencies above a certain level to steeply drop to a zero while not disturbing all frequencies below that level. There is also a *high-pass filter* which does the opposite and attenuates frequencies below a certain level.

The audio recording in my example had a lot of noise typical of old gramophone recordings. When the low-pass filter was applied, the noise disappeared. At that time, I did not know much about FFmpeg filters. If I did, I could have fixed the audio in just one step.

```
ffmpeg -i Stopmotion-hot-wheels.mp4 \
       -filter:a "lowpass=frequency=1000" \
       -codec:v copy \
       Stopmotion-hot-wheels-audio-passed-low.mp4
```

The default option in Audacity was 1000 Hz for the frequency and 6 dB per octave for the roll-off. The roll-off specifies how steeply the frequencies are attenuated. The `lowpass` filter can apply a 3 dB roll-off if you set its `poles` option to 1. The default 2 applies a 6 dB roll-off, and I did not have to explicitly specify it in the above command.

Summary

In this chapter, you learned how to perform several tasks with audio content. You may find it helpful to initially use Audacity to understand audio problems. As you get more familiar with what ails audio content, you can rely on FFmpeg entirely. FFmpeg has a ton of audio filters, and this chapter used just a few of them. Check the FFmpeg documentation on audio filters, and you will find more exciting things you can do with audio.

CHAPTER 9

All About Subtitles

In this chapter, you will learn to perform several tasks related to subtitles. Subtitles are dialogs that are displayed as text on a video. The subtitles may be burned into the video or be available as a separate content stream in the multimedia file. In case of the former, the subtitles cannot be turned off as they have become part of the video. In case of the latter, the subtitles can be turned on or off using a remote button or by selecting an onscreen menu option.

Videos on streaming media, optical media, and broadcast TV can have subtitles in multiple languages. Some websites maintain a crowd-sourced library of subtitles (in multiple languages) of a wide variety of movies, popular and obscure. Several video-hosting sites also display subtitles. They do not let you download subtitles separate from the video. However, there are some other websites that will fetch the subtitles if you give them the address where the original video is hosted.

Subtitles are available in many formats. Subrip (.srt) files are the most popular. Advanced Substation Alpha (.ass or .ssa) is very versatile. WebVTT (Web Video Text Tracks Format) is used by browsers for online videos. TTML is used by the broadcast industry and online applications. DVDs use `.dvdsub` files.

I prefer SSA because I can specify a custom display font with it. For use with FFmpeg, subtitles should be a stream in a media file or an external text file. Subtitles that are already burned into a video (not as a separate stream) cannot be processed by FFmpeg (or rather not covered by this book). However, FFmpeg can be used to burn subtitles permanently on a video.

V. Subhash, *Quick Start Guide to FFmpeg*, https://doi.org/10.1007/978-1-4842-8701-9_9

Add Subtitles to a Video as an Extra Stream

To add a subtitle file to a video, you need to use a subtitle format that is compatible with the video file's container. Or, you should use a suitable encoder that will convert your subtitle file in a format that is supported by the container. The subtitle format for MP4, MOV, and 3GPP containers is known as "MPEG4 Timed Text." You will have to encode your SRT or SSA subtitle files with the encoder `mov_text` for these containers. For the versatile Matroska (MKV) format, you can straightaway use SRT and SSA subtitle files.

Suppose that you have a DVD without subtitles in your favorite language and the DVD seller released a new updated collector's edition DVD that has subtitles in that language. If you were able to download the new subtitles as an SRT file from somewhere, then you can add it to your DVD backup file as an extra stream. If you are saving the DVD as an MKV file, convert the SRT file beforehand to the Substation Alpha (SSA) format to take advantage of the ability of the latter to use a custom font.

```
ffmpeg -i dvd-movie.srt dvd-movie.ass

# Edit the SSA file in some subtitle editor
# and add your custom styles and fonts

ffmpeg -i dvd-movie.ogv -i dvd-movie.ass \
       -map 0:v -map 0:a -map 1:s \
       -c:s mov_text \
       -metadata:s:s:0 language=eng \
       dvd-movie-subtitled.mp4 \
        \
       -map 0:v -map 0:a -map 1:s \
       -codec copy \
       -metadata:s:s:0 language=eng \
       dvd-movie-subtitled.mkv
```

☞ When you add subtitles as an additional stream like this, the viewer can turn them on/off with the device remote or a screen menu option.

Did you notice something else with the above command? I subtitled the movie in two formats (MP4 and MKV) using one command. With the MP4, I had to encode the OGV streams because its codecs are not native to the MP4 container. With the MKV, I could use `-codec copy`. The MKV container supports a wide variety of codecs including those supported by OGV and MP4. If you are backing up DVDs for long-term storage, choose MKV. It is the best.

Permanently Burn Subtitles to a Video

When I was about to publish my first book, I wanted to upload a book-read video in which I read a few pages. I recorded the OGV video using the webcam program *Cheese*, but there were some issues with audio recording. So, I transcribed my narration using another program called *Gnome Subtitles* and saved the subtitles as a Substation Alpha (.ass) file. I did not want to upload the subtitles to the video-hosting sites because they use very tiny fonts. I wanted the subtitles to look bigger and with my own selection of the font. I then decided to use FFmpeg to permanently burn the subtitles on the video. I specified the font and subtitles location on the video in the subtitle file, NOT in the `ffmpeg` command. The SSA format let me do that. Using a filter, I drew a black box behind the subtitles so that they could be easily read against any background.

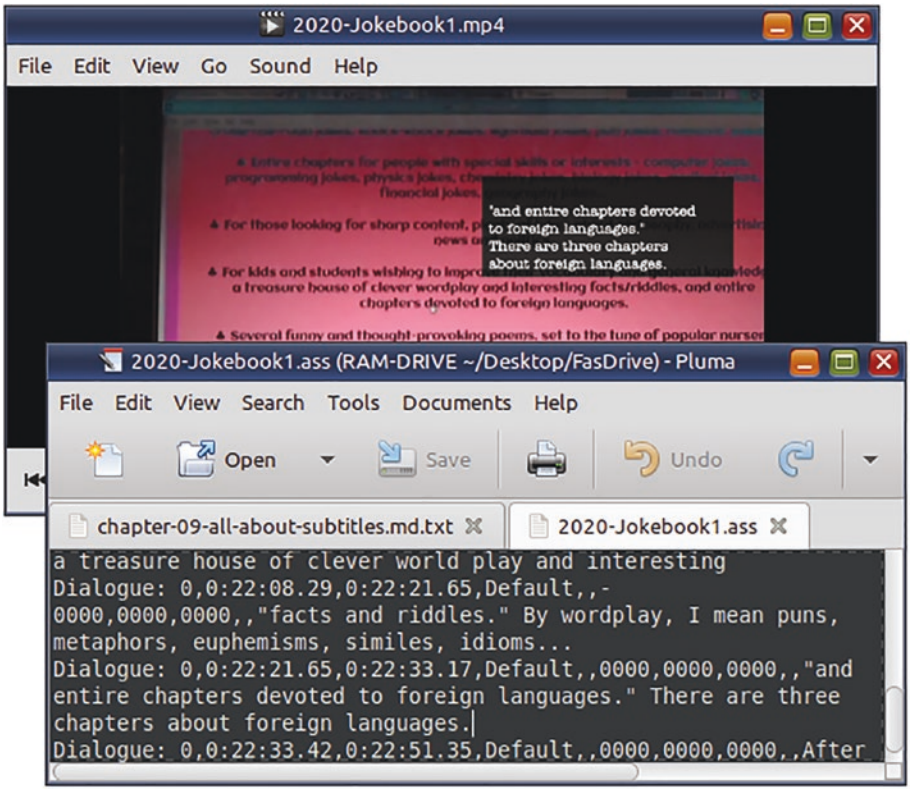


Figure 9-1. *Subtitles burned into a video cannot be turned off with the remote or a menu option*

```
ffmpeg -i 2020-Jokebook1.ogv \
       -filter_complex \
          "drawbox=w=250:h=100:x=360:y=90:color=black@0.7:t=fill,
           subtitles=2020-Jokebook1.ass" \
       -c:v libx264 -r 24 \
       2020-Jokebook1.mp4
```

☞ The `subtitles` filter has a `force_style` option to specify an SSA style for use with a subtitle format (such as SRT) that does not support styles.

☞ The black box was unnecessary. SSA has built-in support for dynamic background boxes, as you will learn later.

Add a Custom Font for Displaying Subtitles of a Video

If I wanted the subtitles in my book-read video to be optional, I could have created an MKV like this:

```
ffmpeg -i 2020-Jokebook1.ogv -i 2020-Jokebook1.ass \
       -codec copy \
       -metadata:s:s:0 language=eng \
       -attach Headline.ttf \
       -metadata:s:t:0 mimetype=application/x-truetype-font \
       2020-Jokebook1.mkv
```

☞ Font embedding increases subtitles portability and toggleability, but support is not universal.

☞ You should place the font file in the current directory or specify its full path.

This command adds the subtitles as an additional stream in the video. It also specifies a custom subtitle display font and embeds that font. On my PC, Totem and VLC display the subtitles with that font. However, my *WDTV HD* media player box, which I used for many years, always played the subtitles with its own built-in font.

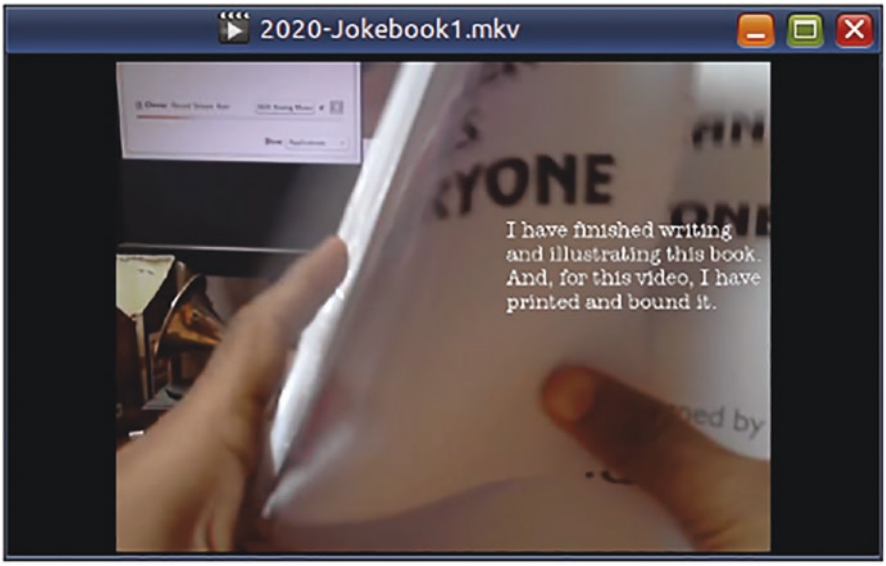


Figure 9-2. *When subtitles are added as a stream, the viewer can turn them on/off using the remote or with a menu option*

About the Substation Alpha (SSA/ASS) Subtitle Format

Although SRT is the popular subtitle format, I prefer the Substation Alpha (.ass or .ssa) because it supports fonts and several other cool features. You can convert SRT to SSA using `ffmpeg`.

```
ffmpeg -i dvd-movie.srt dvd-movie.ass
```

However, I prefer not to do that. I download the SRT file, let it open in a GUI program called *Gnome Subtitles*, and save it as a SSA file. After this, I run a BASH script on the .ass file to change its style statement. The style statement generated by `ffmpeg` and Gnome Subtitles refers to Windows fonts. These fonts are not available in Linux and the resultant subtitles do not look cool. My script uses a better style statement with a font I already have installed in Linux.

`ffmpeg` version:

```
Style: Default,Arial,16,&Hffffff,&Hffffff,&H0,&H0,↵
0,0,0,0,100,100,0,0,1,1,0,2,10,10,10,0
```

Gnome Subtitles version:

```
Style: Default,Tahoma,24,&H00FFFFFF,&H00FFFFFF,↵
&H00FFFFFF,&H00C0C0C0,-1,0,0,0,100,100,0,0.00,↵
1,2,3,2,20,20,20,1
```

My version:

```
Style: Default,Headline,20,&H00FFFFFF,&H006666EE,↵
&H00000000,&HAA00EEEE,-1,-1,0,0,100,100,0,0.00,↵
1,4,0,2,20,20,20,1
```

When I used this style in the book-read video, the subtitles…

```
ffmpeg -y -i 2020-Jokebook1.ogv \
       -i 2020-Jokebook1-shadows.ass \
       -map 0:v -map 0:a -map 1:s \
       -c:v copy -c:a copy -c:s ass \
       -metadata:s:s:0 language=eng \
       -attach Headline.ttf \
       -metadata:s:3 mimetype=application/x-truetype-font \
       2020-Jokebook1-shadows.mkv
```

… look like this:

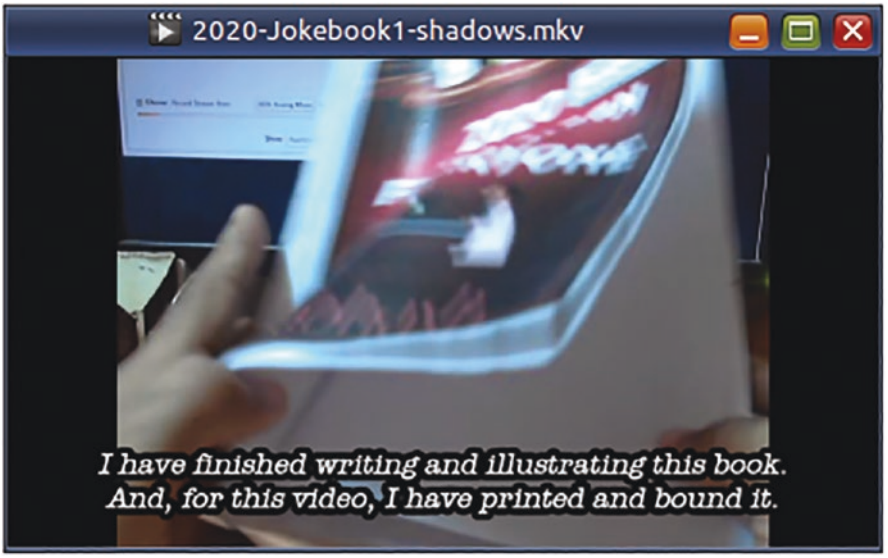


Figure 9-3. *In this video, the subtitles have a text outline. (This eliminated the need to render a black box behind the subtitles using an FFmpeg filter. SSA subtitles support multiple such styles in the same file.) The subtitle shadow has been zeroed*

The specification of the wonderfully useful but screwed-up SSA format is available on the matroska.org website (*Technical Info » Subtitles » SSA*). However, I will risk a description here for the style statement.

```
Style: Name, Fontname, Fontsize, PrimaryColour,
SecondaryColour, OutlineColour, BackColour, Bold,
Italic, Underline, StrikeOut, ScaleX, ScaleY, Spacing,
Angle, BorderStyle, Outline, Shadow, Alignment,
MarginL, MarginR, MarginV, Encoding
```

`Name` refers to a subtitle display style. You can define and use many different styles, not just the `Default`. The colors are in hexadecimal AABBGGRR format. (*Ese*, are they *loco*? No. It is allegedly to help with video-to-text conversion.) `PrimaryColour` is the color of the subtitle text. `OutlineColour` is for the outline of the text. `BackColour` is the color of the shadow behind the text. `SecondaryColour` and `OutlineColour` will be automatically used when timestamps collide. `Bold`, `italic`, et al. are -1 for true and 0 for false. (Yeah, I know. The `bash` shell does the same.) `ScaleX` and `ScaleY` specify magnification (1-100). `Spacing` is additional pixel space between letters. `Angle` is about rotation (0-360) and controlled by `Alignment`. `BorderStyle` uses 1 (outlined and drop-shadowed text), 3 (outline box and shadow box), and 4 (outlined text and drop-shadow box). `Outline` represents the border width (1-4) of the outline or the padding around the text in the outline box. `Shadow` represents the offset (1-4) of the shadow from the text or the space around the text in the shadow box. `Alignment` takes 1 (left), 2 (center), and 3 (right). If you add 4 to them, the subtitle appears at the top of the screen. If you add 8, it goes to the middle. Then, we have margin from the left, right, and bottom edges of the screen. `Encoding` is 0 for ANSI Latin and 1 for Unicode (I think).

To really go bonkers with subtitles, I say we render subtitles with a miasma of colors, location, and tilt.

```
Style: Default,Headline,22,&H6600FFFF,&H006666EE,⏎
&H660000FF,&H220066EE,-1,-1,0,0,100,100,0,25.00,⏎
3,4,4,2,20,20,120,1
```

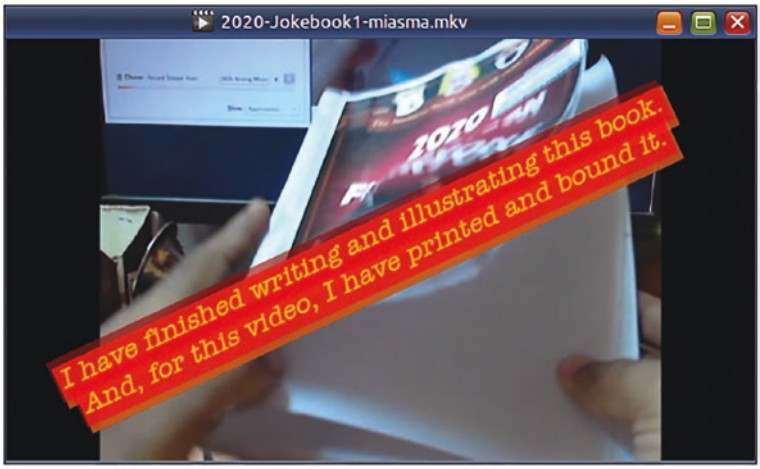


Figure 9-4. *This is truly subtitles gone wild. SSA subtitle format offers the most control and options. There is a yellow shadow to the red outline. Because the colors are translucent, their intersection appears orange*

Add Subtitle Files in Different Languages

When adding multiple subtitles, it is obligatory on your part to specify metadata identifying the language of each output subtitle stream.

Let us pretend that I am trying to corner the French jokebook market and have a French transcript ready as well:

```
# Multi-language subtitled MP4
ffmpeg -i 2020-Jokebook1.ogv \
       -i 2020-Jokebook1-en.ass  -i 2020-Jokebook1-fr.ass \
       -map 0:v -map 0:a -map 1:s -map 2:s \
       -c:s mov_text \
       -metadata:s:s:0 language=eng \
       -metadata:s:s:1 language=fre \
       2020-Jokebook1-subtitled-en-fr.mp4
```

```
# Multi-language subtitled MKV
ffmpeg -i 2020-Jokebook1.ogv \
       -i 2020-Jokebook1-en.ass  -i 2020-Jokebook1-fr.ass \
       -map 0:v -map 0:a -map 1:s -map 2:s \
       -c:v copy -c:a copy -c:s copy \
       -metadata:s:s:0 language=eng \
       -metadata:s:s:1 language=fre \
       2020-Jokebook1-subtitled-en-fr.mkv
```

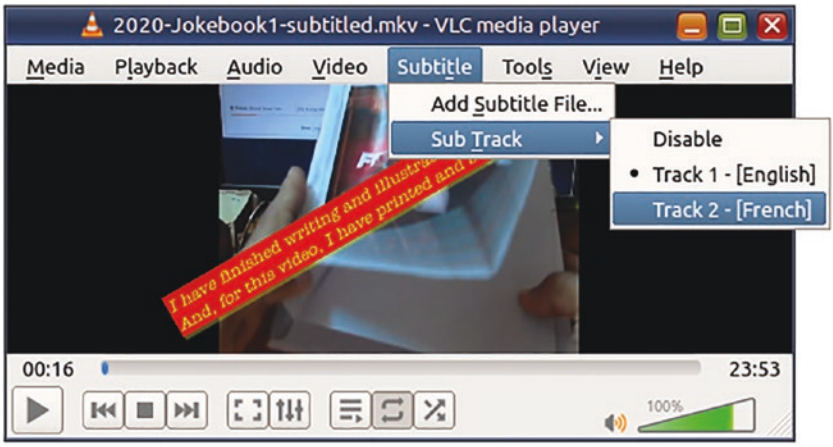


Figure 9-5. *Do not forget to specify metadata for the subtitles*

☞ The codes that you can use for setting the language are further described in Chapter 10.

Extract Subtitles from a Video

Use `ffprobe` to check if a video file has a subtitle stream.

```
ffprobe 2020-Jokebook1-subtitled-en-fr.mkv
```

```
$ ffprobe 2020-Jokebook1-subtitled-en-fr.mkv
Input #0, matroska,webm, from '2020-Jokebook1-subtitled-en-fr.mkv':
  Metadata:
  Duration: 00:23:53.26, start: 0.000000, bitrate: 489 kb/s
    Stream #0:0: Video: theora, yuv420p, 640x360 [SAR 1:1 DAR 16:9], 2
    Metadata:
      DURATION        : 00:23:53.253000000
    Stream #0:1: Audio: vorbis, 44100 Hz, stereo, fltp (default)
    Metadata:
      DURATION        : 00:23:53.259000000
    Stream #0:2(eng): Subtitle: ass (default)
    Metadata:
      DURATION        : 00:23:53.263000000
    Stream #0:3(fre): Subtitle: ass (default)
    Metadata:
      DURATION        : 00:23:53.263000000
```

Figure 9-6. *Use* `ffprobe` *output to identify the subtitle formats and any metadata they might have. und stands for "undetermined"*

If the file has only one subtitle stream, you can extract it using FFmpeg just by specifying the correct extension.

```
ffmpeg -i dvd-movie-subtitled.mp4 \
         dvd-movie-subtitle-default.ass
```

If the video has multiple subtitle streams, you need to specify mapping. The next command saves the second subtitle stream in the input file as an SSA file.

```
ffmpeg -i 2020-Jokebook1-subtitled-en-fr.mkv \
         -map 0:s:1 \
         2020-Jokebook1-subtitle-fr.ass
```

Extract Subtitles from a DVD

The files in a DVD are usually encrypted or obfuscated to prevent bootlegging. There are several free DVD-ripping applications that will decrypt the VOB files and quickly extract subtitle files. Forcing `ffprobe` to find subtitle streams on big VOB files is not worth the trouble.

Summary

Subtitles are available in several formats including SRT, Substation Alpha, and MPEG4 Timed Text. The Substation Alpha is the most versatile subtitle format, and MKV seems to be the best container for it. The style specification for the Substation Alpha format may seem intimidating at first but will be accommodative in customizing subtitles for a variety of use cases.

CHAPTER 10

All About Metadata

In this chapter, you will learn to perform several tasks related to metadata. Metadata means to data about data. Multimedia metadata refers to information such as title, artist, album, subject, genre, year, copyright, producer, software creator, comments, lyrics, and even album art images that are used to describe the video and/or audio content.

An audio or video file can have global metadata (i.e., at the file level) and stream-specific metadata too. You can use `ffprobe` and `ffmpeg -i` commands to display metadata that a file already has. You use the `-metadata` option to add new metadata.

Add Album Art to MP3

You can add several pieces of album art to an MP3 file. However, each image will need to have a unique title and comment metadata. There can be one for front cover, another for the back, and yet another for the inlay art. FFmpeg will treat all album art images as video streams, as if they were single-frame videos.

```
ffmpeg -y \
  -i Uthralikavu-Pooram.mp3 \
  -i Uthralikavu-Pooram-festival-fireworks.png \
  -i Uthralikavu-Pooram-festival-crowds.png \
```


V. Subhash, *Quick Start Guide to FFmpeg*, https://doi.org/10.1007/978-1-4842-8701-9_10

```
-map 0 -map 1 -map 2 \
-metadata:s:1 title="pooram-fireworks.png" \
-metadata:s:1 comment="Cover (front)" \
-metadata:s:2 title="pooram-crowds.png" \
-metadata:s:2 comment="Cover (back)" \
-codec copy \
-f mp3 \
Uthralikavu-Pooram-festival-fireworks.mp3
```

☞ Album art are added as single-frame video streams, not metadata. The metadata you add for album art will apply to the video streams of those images.

There are several options for the `comment` key, as defined in the ID3 tag specification.

```
https://id3.org/id3v2.3.0
```

There is no uniform implementation among media players. When there are more than one album art images, `ffplay` chooses the first cover image that is mapped. Some other players follow a different pecking order.

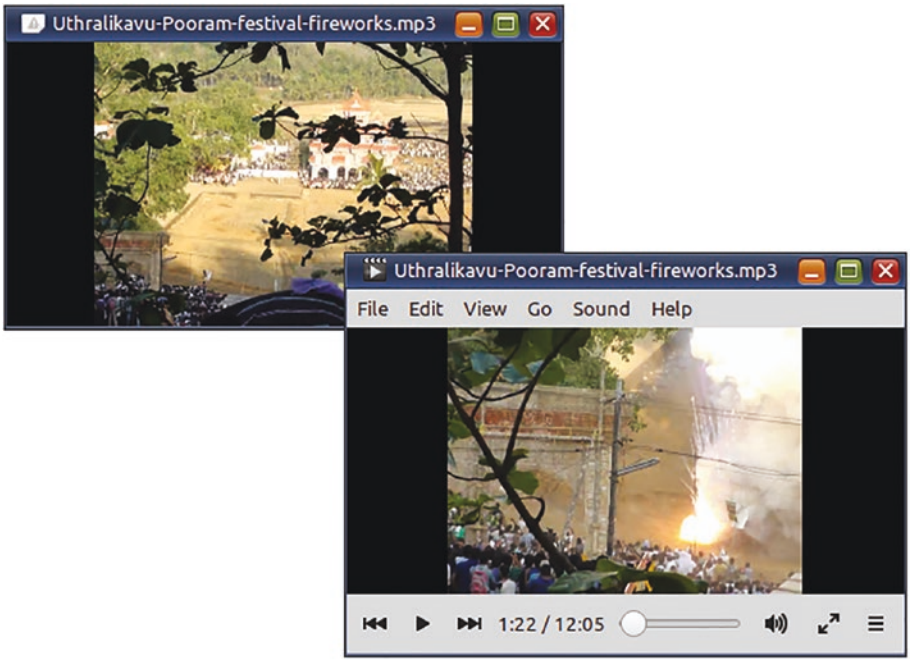


Figure 10-1. *The album art displayed by different media players for the same MP3 file can be different*

Set MP3 Tags

How do I add metadata to an MP3 file?

```
ffmpeg -y -i Uthralikavu-Pooram-festival-fireworks.mp3 \
       -map 0 \
       -metadata title="Uthralikavu Pooram Festival" \
       -metadata artist="V. Subhash" \
       -metadata \
          subject="Fireworks and crowds" \
       -metadata album="Pooram festival fireworks" \
       -metadata date="2013-12-26" \
       -metadata genre="Event" \
```

```
        -metadata comment="Best outdoor event I ever attended" \
        -metadata \
           copyright="© 2013 V. Subhash. All rights reserved" \
        -id3v2_version 3 \
        -codec copy \
        Kerala-Uthralikavu-Pooram-festival-fireworks.mp3
```

☞ MP3 tags metadata get added at the global level. They are not stream-specific.

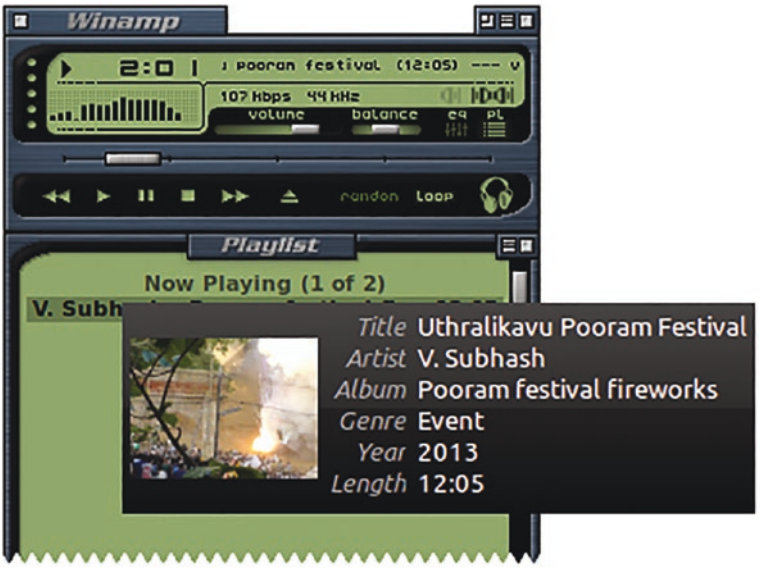


Figure 10-2. *Media player support for MP3 tags may be buggy or not 100%. Do not break your head just because some tags do not get displayed by a media player*

Export Metadata

You can export metadata to a text file using the `-f ffmetadata` option.

```
ffmpeg -i Kerala-Uthralikavu-Pooram-festival-fireworks.mp3 \
       -f ffmetadata \
       mp3-meta.txt
```

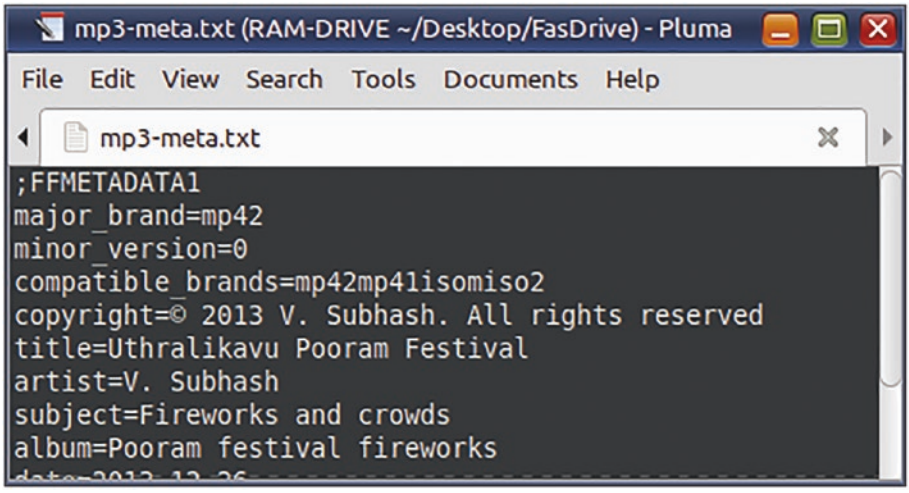


Figure 10-3. *ffmpeg exported this text file containing name-value pairs representing the metadata of an MP3 file*

Import Metadata

Let us imagine that I modified the metadata in the text file (from the previous section) using a text editor. Now, I want the updated metadata to be imported back into the audio file. How can I do it?

```
ffmpeg -y \
 -i Kerala-Uthralikavu-Pooram-festival-fireworks.mp3 \
 -i mp3-meta-modified.txt \
 -codec copy \
 -map_metadata 1 \
 Kerala-Uthralikavu-Pooram.mp3
```

Here, `-map_metadata 1` refers to the second input file, that is, the modified metadata file. (`-map_metadata 0` would have simply copied the metadata from the first input file, that is, the MP3 file. We did not want that.)

Extract Album Art

You downloaded an MP3 and you like the album art? If the audio file has only one album art, you can extract the image easily.

```
ffmpeg -i Blobfish.mp3 blobfish-album-art.png
```

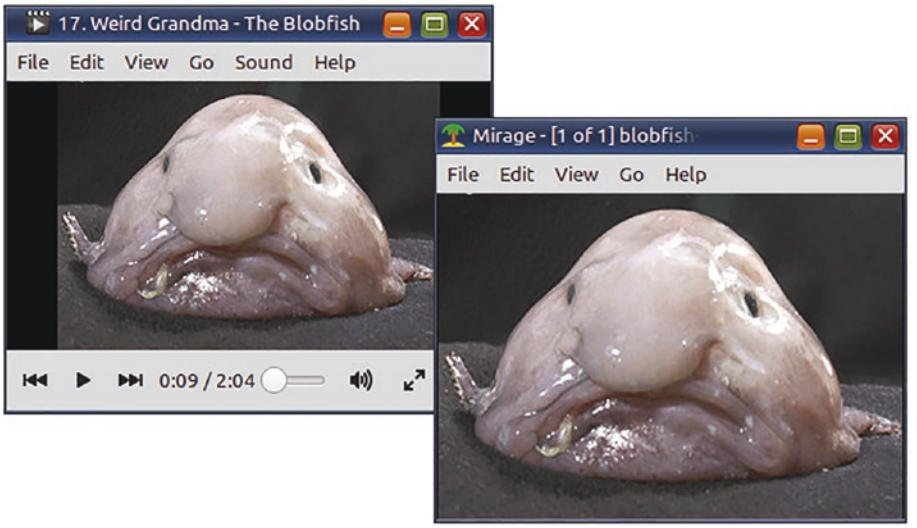


Figure 10-4. *An MP3 audio file and the album art extracted from it*

If there are more than one album art, you need to check the ffprobe output and then extract the album art using a map.

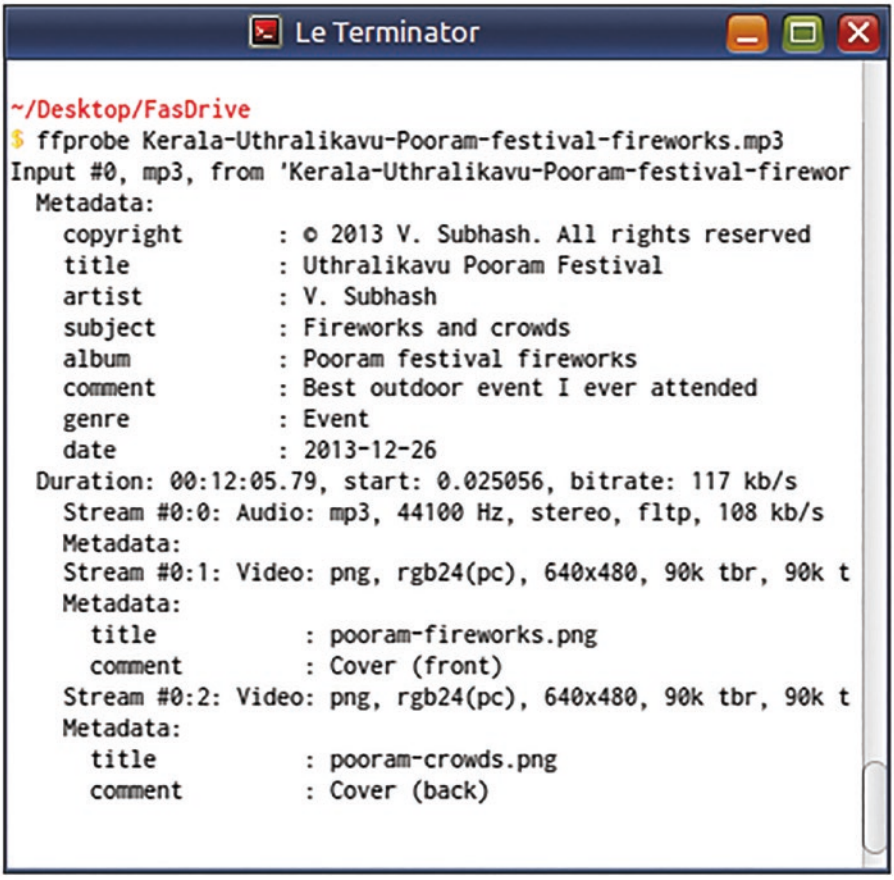


Figure 10-5. *This ffprobe output shows the index of the streams containing the album art images*

The crowds image is identified as a video stream with index `0:2` (third among all streams). To extract it, I should use the map `0:2`. To be safer, I refer to it as `0:v:1` (second video stream).

```
ffmpeg -i Kerala-Uthralikavu-Pooram-festival-fireworks.mp3 \
        -map 0:v:1 \
        crowds.png
```

Remove All Metadata

When working on an earlier chapter, I found that the Mate Screenshot app was unable to work with the video of the sign-language translator. The app names its screenshot after the title of the subject window. I noted that this video had a URL displayed in the title of the video player window. The URL came from the title metadata of the video. Because the Linux file system does not allow a file name to include a URL (because of the backslash and other illegal characters), the screenshot app may have been unable to save the image to file. When I removed the metadata, I realized that my hunch was right and I was able to take the screenshots from the metadata-free video.

To remove the metadata, I pretended to import metadata from a nonexistent input file (with index `-1`).

```
ffmpeg -i "Sign_Language_-_How_To_Vote.mp4" \
       -codec copy \
       -map_metadata -1 \
       how-to-vote.mp4
```

I have had portable media players that do not play MP3 files if they have album art. Album art cannot be removed as metadata because they are encoded as video streams. So, I use `-codec copy` and specify a `-map` for the audio stream. By omitting video streams, the output file will not have any album art.

```
ffmpeg -i Kerala-Uthralikavu-Pooram.mp3 \
       -map 0:a \
       -codec copy \
       pooram.mp3
# You can also use -vn instead of the -map option
```

Set Language Metadata for Audio Streams

Let us imagine that I created audio instructions in English, Malayalam, and Tamil for this DIY electronics video. While media players could switch between the language tracks, they would have assigned generic or confusing names to them.

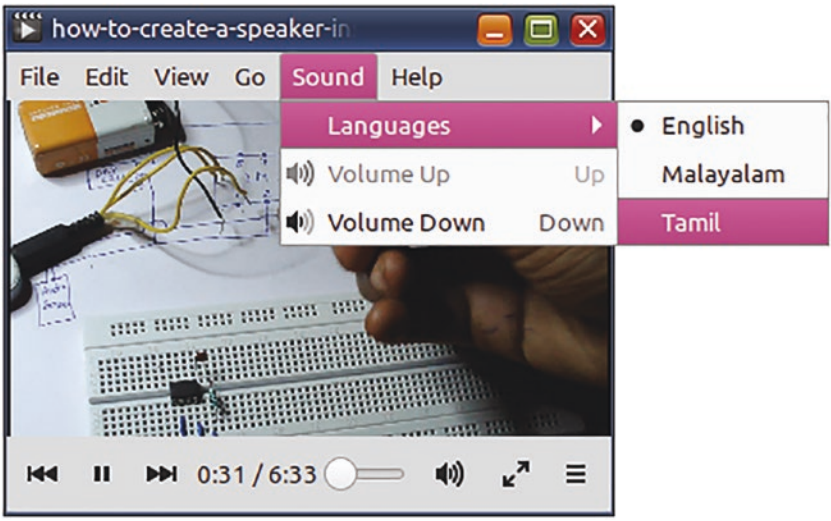


Figure 10-6. *This video has audio tracks in three languages. The metadata for the audio streams helps identify the languages*

The following command sets the language names using ISO codes and makes the menus a lot more informative.

```
ffmpeg -i how-to-create-a-speaker-instructions.mp4  \
       -map 0 \
       -metadata:s:a:0 language=eng \
       -metadata:s:a:1 language=mal \
       -metadata:s:a:2 language=tam \
       -codec copy \
       how-to-create-a-speaker-instructions-multilang.mp4
```

`map 0` includes all streams in the first input file (#0), that is, including the video stream and the three audio streams. (If not used, there will be just one video stream and one audio stream in the output file.) `-metadata:s:` is used to set metadata for a stream, not a subtitle.

☞ Apart from `s` identifier for streams, FFmpeg uses identifiers `p` and `c` for DVD programs and chapters of the VOB file container. These are not covered by this book.

`-metadata:s:a` is used to set metadata for an audio stream specified by its index. `language` is the metadata key, and what follows after the `=` sign is the value in the metadata key-value pair. `-codec copy` ensures that the streams are not re-encoded – only the metadata is added.

The three-letter language codes (such as `eng, mal,` and `tam`) are specified in the **ISO 639-2** standard. Although the standard allows codes for exceptional situations (`mis` for "uncoded languages," `mul` for "multiple languages," `qaa-qtz` for "reserved for local use," `und` for "undetermined," and `zxx` for "no linguistic content" or "not applicable"), many software and hardware remain ignorant of them.

`www.loc.gov/standards/iso639-2/php/code_list.php`

Summary

In this chapter, you learned to use ffmpeg to easily add, examine, edit, export, import, and remove metadata. Metadata can be specified at the container level (global) and for individual streams. This information can greatly enrich the experience with media players. In their absence, media players will try to make guesses and/or frustrate you with generic or wrong interface choices. Media formats and software/hardware applications may be picky and choosy about the kind of metadata they support.

With the end of this chapter, all that remains is a set of tips and tricks that could not be accommodated anywhere else.

Index

A

V. Subhash, *Quick Start Guide to FFmpeg*, https://doi.org/10.1007/978-1-4842-8701-9

B

C

D

E

F

G

H

I, J, K

L

M

N

O

P, Q, R

S

T, U

V, W, X, Y, Z