Aurora Development Journal

Sacha El Masry

May 29, 2017

Contents

Abstract

Multimedia has been ubiquitous for nearly thirty years, it is the most consumed form of information on the internet, yet manipulation tools are balkanised, making it difficult to manipulate these assets. Video is produced, manipulated and viewed on an incredible range of excellent and versatile tools, yet no one single tool provides full and relevant control over important aspects of the delivery format. Chiefly, fine control over encoding settings, size, compression is either absent or very primitive. Even worse, despite exhortations to attach metadata to all assets, most video editing and manipulation software doesn't provide any control, and when it does, the result is inconsistent with established standards or media players, making it useless.

Aurora is my attempt to address these two concerns head on; to make delivery video files more pliable with fine control, as required for a range of uses, from hosting on an independent web-site to various social video-sharing hubs. Secondly, and even more importantly, to discover working metadata fields, by format, and to provide a way to consistently and reliably embed metadata into these delivery files, to promote archival, author rights management and discoverability.

1 Introduction and Background

While I have need for for all aspects of multimedia management, my most pressing need is video manipulation. I identify two glaring problems with the situation, motivatng this development, and it is not the dreaded *not invented* *here* syndrome[1] driving me.

First, my problem is in fine-tuning final deliverable video files. I admit that making videos now is incredibly easy, and that we have a range of powerful video editors available at all prices, as well as open-source: Final Cut Pro X, DaVinci Resolve, Lightworks, Blender and KDEnlive are the most important that I've used. In my experience, they offer a good—with the exception of Resolve, not excellent—choice of target video formats, but, apart from Blender, very little control over fine-tuning the output file is supported by most editing software. Even when there is a semblance of control provided, it either doesn't work, or permits the production of incompatible files—files which will not play on a number of mainstream media players.

Before I come across as a nit-picker (which I certainly am), once a video producer's needs surpass entry-level video production, there are many good and valid reasons why the producer desires control over the delivery format. For one thing, the receiving party may have strict requirements and specifications; a target platform may restrict uploads to a maximum size; the videographer, editor, colourist or VFX technician each have their own reasons to maximise multiple constraints, based on their knowledge of the source material and its target purpose. In all cases, we're dealing with constrained resources, and we may desire to affect dozens of attributes to tweak final or intermediate delivery files, in what is basically a *constraint optimisation* exercise.

Beyond merely obsessive control over having the most tuneable attributes, we need flexibility in our output formats, for reasons and uses that are simply unquantifiable. Then, once we reach the right output format and settings, if there is the slightest expectation that this action needs to be repeated more than once—per project or across different projects—we want these choices to be memorised by our system, making these actions easily repeatable—as well as reproducible, for scientific needs. For example, in 2017, advice for publishing video to an independent website is to prepare three output formats, covering multiple consuming browsers, older versions of these browsers, and browsers on different platforms. These formats are: MP4[2], WebM[3] and OGV[4]. It's easy to use any of a range of software to create one or two of these (rarely all three), without resorting to the command-line and FFmpeg[5], but many of these graphical tools actually build on FFmpeg to begin with.

FFmpeg is a monster project in the open-source video world; it's aim is to be the leading framework to stream and manipulate almost every format ever created, from the most obscure to the cutting edge. Being such a well-maintained and wide-ranging tool makes it the software that many video manipulation packages—or FFmpeg front-ends, really—are built on. While FFmpeg itself is excellent, many of these front-ends are not. Among the best is HandBrake[6], which does a fantastic job of exposing many of FFmpeg's options, but provides a free-form way to specify absolutely any other options. The problem is that HandBrake was designed to provide a way to convert DVDs and other heavy video formats (AVI, DV, etc.) to more compressed, and possibly resized H.264-formatted videos, for watching on a laptop or other small device. Why is this a problem? It has never shed this identity, so using my prior example of transcoding a master video to the three formats desired for websites, is difficult, only because HandBrake doesn't expose WebM and OGV as destination formats—it does, but it names the files with a less-supported file extension, requiring a further manual step.

My second problem seemed trivial (and I still can't believe it isn't), in writing metadata to video files. In all aspects of Digital Asset Management, creators, marketers and archivists know that metadata is necessary to: (a) assert the author's rights; (b) help in discovery of the asset, either for use, by marketers, or for consumption by end users; and (c) to keep the asset's history enshrined for future archeology. Coming from an amateur photography fascination, and later an audio transcoding and collection one, this seemed like a non-problem at best, or solved problem at worst. My experience in the above two fields, and all that has ever been written about them means we are all familiar (at least in passing) with their metadata, and that all software can handle it. In photography, we have

[1]Not invented here is a stance adopted by cultures that avoid using or buying already existing products or knowledge because of their external origins and costs (Wikipedia, `https://en.wikipedia.org/wiki/Not_invented_here`)

[2]MPEG-4 Part 14 (Wikipedia, `https://en.wikipedia.org/wiki/MPEG-4_Part_14#Metadata`)

[3]WebM is a video file format, primarily intended to offer a royalty-free alternative to use in HTML5 video (Wikipedia, `https://en.wikipedia.org/wiki/WebM`)

[4]Ogg is a free, open container format maintained by the Xiph.org foundation (Wikipedia, `https://en.wikipedia.org/wiki/Ogg`)

[5]FFmpeg: "A complete cross-platform solution to record, convert and stream audio and video" (`http://ffmpeg.org/`)

[6]HandBrake is a tool for converting video from nearly any format to a selection of modern, widely supported codecs (`https://handbrake.fr/`)

IPTC[7], XMP[8], Exif[9] and Dublin Core[10], and every conceivable manipulation, viewing and album software reads and handles one or more of these standards. In audio, we're familiar with the ID3 tag standard in all its versions, and all respectable music editors, manipulators, players and media managers can read and write ID3[11] tags. It's because of this that we can share photos (as well as other images) and audio files, and know their title, description, date of capture, genre and more, and we can do this repeatably, portably across platforms and players and the general media. On the internet, third-party websites and sharing platforms may strip this data, but that's a different story.

Not so easily in video; a search for metadata in images and audio returns countless results either suggesting the stripping of metadata from the file for faster web page loading, or best practices for storing metadata. A similar search for video is strangely silent on the issue. As there are many acceptable standards in use, each of which accept a different standard of metadata, it would be understandable that there were multiple opinions expressed on which subset of metadata is safest across formats and players, along with a lively discussion around the issue. Instead there is very little quality information available, and the issue as a whole appears to have been shunned and buried as irrelevant. FFmpeg front-ends on the whole ignore metadata altogether, witness HandBrake; others merely give it a nod, using only the most common half-dozen fields for basic use. Video editing software is the worst, in either plainly refusing to save metadata with the rendered video, or exposing an insultingly small number of fields, three in the case of Final Cut Pro X. Finally, Digital Asset Management software, in the main, stores extended metadata externally, in its own database; when these video files are shared (i.e. sent outside the confines of the DAM), it is unclear whether any of this is written to the video file itself. Archivally speaking, the answer is probably not, instead relying on sending the video exactly as it was ingested, or supplying a plain-text (or XML) sidecar file containing the metadata that was saved for the video.

FFmpeg itself permits the user to call any number of `-metadata TAG=value` arguments, but the range of acceptable (by the destination format) tags is not documented at all. Even when FFmpeg is pressed into service to transcode a video and inject commonly used tags, an Exiftool[12] inspection demonstrates they're not saved in the file. Is this a case of FFmpeg not writing some tags? Or is it the file format that doesn't accept saving of the tag? This situation is exasperating, but what's worse is that the next popular format relies on a completely different metadata standard which either uses similar (but slightly different) tags, or tags with completely different names.

Reading FFmpeg's source code would quickly reveal which tags are written for which output format, and this is easy to test, but as a process it's limited to the current version (plausibly linked to the target platform), as well as flags used in compiling the binary, making it likely that it won't be informative. Reading the source code, means relying on it, requiring tracking the project, and keeping up with its changes. In my search, I'd like to discover a more generic and portable way to detect which tags—from each format, from its standards' documents—are successfully written and subsequently read back, autonomously. Further, I want to come up with a way of dealing with differing tag names for effectively the same information, across formats.

2 Desiderata

2.1 Metadata

I want to discover a way to test what meta tags can be applied, and reliably read back from a file. FFmpeg is one approach, but I have no way of knowing if it's the best tool for the job, and will pit it's abilities against Exiftool, hoping that they will either perform metadata writing identically, or that one or the other will be clearly superior, so that I can concentrate my efforts on it.

[7]The Inormation Interchange Model (IIM) is a file structure and set of metadata attributes, applicable to images (Wikipedia, `https://en.wikipedia.org/wiki/IPTC_Information_Interchange_Model`)

[8]The Extensible Metadata Platform (XMP) is an ISO standard, created by Adobe Systems Inc. of standardised metadata for digital documents (Wikipedia, `https://en.wikipedia.org/wiki/Extensible_Metadata_Platform`)

[9]Excchangeable image file format, is a standard specifying the formats for images, sound and ancillary tags used by digital cameras and other systems handling image and sound files recorded by digital cameras (Wikipedia, `https://en.wikipedia.org/wiki/Exif`)

[10]The Dublin Core Schema is a small set of vocabulary terms that can be used to describe web resources, including video, images, CDs and other works (Wikipedia, `https://en.wikipedia.org/wiki/Dublin_Core`)

[11]ID3 is a metadata container designed for use with the MP3 audio file format, as timed metadata in HTTP live streaming, and later widely used in AIFF and WAV uncompressed audio formats, and the MP4 compressed audio format (Wikipedia, `https://en.wikipedia.org/wiki/ID3`)

[12]Exiftool is a platform-independent tool for reading, writing and editing meta information in a wide variety of files (`http://www.sno.phy.queensu.ca/~phil/exiftool/`)

I will need a starting point to use as a gauge of these tools' effectiveness, and to that end I expect to use published standards, such as they are. The standards I have discovered to date are:

1. Extensible Metadata Platform (XMP) :: Adobe's labelling technology for embedding data about a file into the file itself. MP4 files contain metadata defined by the format itself and in addition can contain XMP metadata as documented (`http://www.adobe.com/devnet/xmp.html`). The format is defined in the ISO/IEC 14496-12:2015 standard

2. Matroska :: WebM was designed to initially support a subset of the Matroska specification. WebM global metadata is described in (`http://wiki.webmproject.org/webm-metadata/global-metadata`).

3. Support for Theora and FLAC comments in Ogg containers is very limited.

Given these specifications, all tags relevant to the format should be written to a test file, provided with controlled values, by both tools, and then read back to ensure the tags were actually written, that they are readable and contain valid data, as supplied in the writing step.

Further, since FFmpeg is called on the command-line, and some systems have a maximum character length they allow in one command, for the sake of portability, it will be prudent constructing an external file with all the metadata saved in it. FFmpeg is designed to read an external metadata file, when supplied.

2.2 External configuration

Hard-coding configuration is hardly the way to progress. While the prototype manages to work this way, this has got to change quickly, permitting consumers of the library/application to define their own configuration settings in preferably plain-text files. This depends on reading environment variables and interacting with the operating system (OS) and the underlying file system.

Initially, the configuration file can be defined in Lisp S-expressions, but once this project is proven feasible, that must change to a format more generally used and understood, such as YAML, or Windows `.ini` style of configuration.

2.3 Argument construction

Construction of the command-line argument list to pass to FFmpeg is initially handled by concatenating list items into a long string. It works, but isn't pretty, and it provides no smart argument composition, as may be required by FFmpeg. In any case, it's not idiomatic to a Lisp developer, so all possible arguments that FFmpeg supports must be made available for consumption, in a Lispy way.

2.4 Wider category of media

At first video, but why stop there? I handle a wide range of digital media files, each with a different metadata structure. As I'm producing a significant quantity of imagery, video and audio for the web, Aurora should remain somewhat media-agnostic to enable expansion to these data types in the future.

2.5 Interface

Aurora is developed in Common Lisp, as a library that can be used in other projects. All functionality can be reached from the Lisp REPL. Once this project matures and proves itself not only capable, but useful (and used), I will progress to creating interfaces for it.

On the one hand, a command-line interface may be interesting, but I question its usefulness, so I won't concentrate on developing one, at least in the short-term.

The first logical external interface is a simple web front-end. This will provide enough decoupling between the front and back-ends, that if I decide that a stand-alone graphical (portable) interface is needed, it will by then be easy to develop.

3 Development Methodology

3.1 Literate programming

I started developing this project in a simple, classical way, and reached a prototype able to read a list of possible meta tags, constructing a function to expect exactly those tags as keywords, with basic conversion profiles stored in variables within the source code, no possibility of external configuration and no way to easily specify FFmpeg options in a one-time manner, i.e. outside a stored profile. To me, that method can only take me so far, and I feel I've reached a point where I need to annotate conflicting thoughts and construct a record of information obtained about the problem at hand. I find it too stifling to keep all the thoughts in my head, and I need a structured way to explore and define ways forward, a roadmap. With it, I desire a running journal of problems encountered, options available, decisions made, and what led to those choices, rejecting other options.

I find Literate programming provides this annotated fourth dimension; even though source control provides the timeline of the codes' progress, it does not easily demonstrate the reasoning, perceived tradeoffs, and otherwise cryptic code arrived at through optimisation.

The Literate programming method also appeals, as this document is the source for both the project's complete source code—through tangling—and the typeset, readable version of this development document—using the export process.

3.2 Testing

Testing will play an important part in this project, but I will not practice test-driven development (TDD). While it has its place in highly structured and specified projects, where an API has been defined in full, it has no place in experimentation and the process of discovery, where the API is constantly in flux. In this instance it's an evil, much like premature optimisation. The cost of having no up-front tests is variable reliability, and lots of repeated manual testing instead, with all the associated problems.

As features mature, and I find myself changing and re-factoring less, I will introduce tests for precisely those features, using intuitive tests at first, followed by those that reliably reproduce bugs and ill behaviour.

3.3 Versioning

I don't want to get bogged down in versioning hell, and expect to use simple, OpenBSD-style versioning in the form of a single decimal point, i.e. `x.x`. This is a simple iterator, not denoting any specific major/minor version, development, alpha/beta release, bug-fix, or any other meaning.

3.4 Licensing

FFmpeg is licensed under the GPL, and if used directly, as a library, require this project to be GPL-licensed too. I want to keep the project permissive under the BSD license, which means that I will interface with FFmpeg through the command-line, and not directly.

4 Project Framework

4.1 Human-readable project files

Every project needs a few basic—plain-text—files to quickly inform interested developers and users about its goals, methodology, license, as well as quick installation and usage instructions.

These files are:

`README.markdown` used mainly by Github

`README.org` used for more detail, a richer level of expression and the ability of transformation into a greater range of target formats and uses (as demonstrated by this, Literate development document)

`LICENSE` document spelling out this project's licensing terms

Aurora is free software, licensed under the permissive BSD license, the terms of which are laid out in the LICENSE file, in the root directory of the project's source code.

4.2 Project definition

The heart of every project, source code here implies *all* code—in any language—needed to set up the project and run-time environment, to run the program, and to tear down after its completion.

In Common Lisp, this begins with the system definitions, including the definitions of its shadow testing environment:

`aurora.asd` this is the ASDF system definition file, including all information and dependencies of the system

`aurora-test.asd` as above, but for the testing environment only

```
#|
  This file is a part of aurora project.
  Copyright (c) 2017 Sacha El Masry (sacha@rojoynegroclub.com)
|#

#|
  Aurora is a video file resizer and metadata tagger.

  Author: Sacha El Masry (sacha@rojoynegroclub.com)
|#

(in-package :cl-user)
(defpackage aurora-asd
  (:use :cl :asdf))
(in-package :aurora-asd)

(defsystem aurora
  :version ”0.2”
  :author ”Sacha El Masry”
  :license ”BSD”
  :depends-on (:alexandria
               :cl-strings)
  :components ((:module ”src”
                :components
                ((:file ”filesystem-interface”)
                 (:file ”aurora”))))
  :description ”Aurora is a video file resizer and metadata tagger.”
  :long-description
  #.(with-open-file (stream (merge-pathnames
                             #p”README.markdown”
                             (or *load-pathname* *compile-file-pathname*))
                            :if-does-not-exist nil
                            :direction :input)
      (when stream
        (let ((seq (make-array (file-length stream)
                               :element-type ’character
                               :fill-pointer t)))
          (setf (fill-pointer seq) (read-sequence seq stream))
          seq)))
  :in-order-to ((test-op (test-op aurora-test))))

#|
This file is a part of aurora project.
Copyright (c) 2017 Sacha El Masry (sacha@rojoynegroclub.com)
|#

(in-package :cl-user)
(defpackage aurora-test-asd
  (:use :cl :asdf))
(in-package :aurora-test-asd)

(defsystem aurora-test
  :author ”Sacha El Masry”
  :license ”BSD”
  :depends-on (:aurora
               :prove)
  :components ((:module ”t”
                :components
                ((:test-file ”aurora”))))
  :description ”Test system for aurora”

  :defsystem-depends-on (:prove-asdf)
  :perform (test-op :after (op c)
                    (funcall (intern #.(string :run-test-system) :prove-asdf) c)
                    (asdf:clear-system c)))
```

The initial source-code file is `aurora.lisp`, stored in the `src` directory, and its counterpart file for testing is t/aurora.lisp.

```
(in-package :cl-user)
(defpackage aurora
  (:use :cl :aurora/filesystem-interface)
  (:import-from :uiop
   :run-program)
  (:import-from :alexandria
   :when-let)
  (:import-from :cl-strings
   :join
                :split
   :clean
                :camel-case
   :kebab-case
                :snake-case))
(in-package :aurora)
```

```
(in-package :cl-user)
(defpackage aurora-test
  (:use :cl
   :aurora
        :prove))
(in-package :aurora-test)

;; NOTE: To run this test file, execute '(asdf:test-system :aurora)' in your Lisp.

(plan nil)

;; blah blah blah.

(finalize)
```

Starting with a working prototype, I've already refactored a multitude of OS and filesystem interface functions into its own source file, `src/filesystem-interface.lisp`.

```
(in-package :cl-user)
(defpackage :aurora/filesystem-interface
  (:nicknames :afi)
  (:use :cl)
  (:import-from :uiop
   :ensure-pathname
                :probe-file*
   :delete-file-if-exists
                :truename*
   :directory-exists-p
                :file-exists-p
   :ensure-pathname
                :ensure-directory-pathname
   :resolve-location
                :resolve-absolute-location
   :pathname-directory-pathname
                :absolute-pathname-p
   :relative-pathname-p
```

```
                :file-pathname-p
   :split-name-type)
  (:import-from :alexandria
   :when-let)
  (:import-from :cl-strings
   :join
                :split
   :clean
                :camel-case
   :kebab-case
                :snake-case)
  (:export #:absolute-directory-from-path
           #:construct-directory
           #:split-path
           #:construct-destination-directory
           #:construct-file-name
           #:construct-destination-path))
(in-package :aurora/filesystem-interface)
```

4.3 Other computer-readable project files

Apart from the obviously necessary source-code, a few other computer-readable files are often necessary for the smooth development, testing or distribution of the software.

The most pressing are specific instructions for source-control systems (SCM), in the form of a list of custom directives. Generally, SCM systems need to be told which files or file patterns to ignore. I currently use Fossil SCM, but will target Git as well, as it is a very widely used SCM. The files needed are:

`.gitignore` the Git ignore file

`.fossil-settings/ignore-glob` the Fossil SCM ignore file

These files are intended to tell their respective SCM to ignore dot-files, backup and temporary files produced by Emacs and the like, Lisp image files (`.fasl`, `.*fsl`), and the `db/` directory, as user-specific information **must not** ever be committed and stored in a version-controlled system. These files must be viewed as private and confidential and must be backed up and stored by the user or administrator of the system specifically, and are not in any way part of the project.

5 Filesystem and Operating System Interface

```
#|
FILE MANAGEMENT

This section builds utilities to help with file management, parsing and generating
path names.
|#
(defun absolute-directory-from-path (path &key exists-p)
  "Return directory PATH points to."
  (when (and path
             (or (pathnamep path)
                 (and (stringp path)
                      (not (zerop (length path))))))
    (if exists-p
        (probe-file*
         (nth-value 0 (split-path path)))
```

```lisp
        (nth-value 0 (split-path path)))))

(defun construct-directory (absolute-directory relative-directory)
  ”Construct and return a valid new directory by adding a RELATIVE-DIRECTORY
component to an ABSOLUTE-DIRECTORY.”
  (make-pathname
   :directory (append
               (pathname-directory absolute-directory)
               (list relative-directory))))

(defun split-path (path)
  ”Return PATH split into its constituent components as return values:
- directory
- full file name
- file name
- file type.

Adapted from Weitz, E. (Common Lisp Recipes, 2016, p.446)”
  (when (and path
             (or (pathnamep path)
                 (and (stringp path)
                      (not (zerop (length path))))))
    (let ((file-name (file-namestring path)))
      (if (and file-name (plusp (length file-name)))
          (multiple-value-bind (name type)
              (split-name-type file-name)
            (values (directory-namestring path)
                    file-name name type))
          (values (directory-namestring path) file-name NIL NIL)))))

(defun construct-destination-directory
    (&key source-path destination-directory ensure-directory-exists-p)
  ”Return a fully constructed and valid destination directory, given SOURCE-PATH
and DESTINATION-DIRECTORY components.

Scenarios:

DESTINATION-DIRECTORY is provided, and it is an absolute path, then construct
the destination directory else return an error. This is the default desired
behaviour.

Return: ’DESTINATION-DIRECTORY’

SOURCE-PATH is provided and is (or it’s parent directory) is an absolute path,
construct a destination path from it, else return an error.

Return: ’parent directory of SOURCE-PATH’

If SOURCE-PATH is provided with an additional, optional, DESTINATION-DIRECTORY,
and DESTINATION-DIRECTORY is a relative path, then construct a destination path
including this.

Return: ’parent directory of SOURCE-PATH/DESTINATION-DIRECTORY’
```

```
Optionally, check that the constructed destination directory exists.”
  (cond ((and destination-directory
              (absolute-pathname-p destination-directory))
         (if ensure-directory-exists-p
             (absolute-directory-from-path destination-directory :exists-p t)
             (absolute-directory-from-path destination-directory)))
        ((and source-path
              (absolute-pathname-p source-path)
              (if ensure-directory-exists-p
                  (absolute-directory-from-path source-path :exists-p t)
                  t))
         (if destination-directory
             (and
              (relative-pathname-p destination-directory)
              (when-let ((constructed-directory
                          (construct-directory
                           (absolute-directory-from-path source-path)
                           destination-directory)))
                (if ensure-directory-exists-p
                    (absolute-directory-from-path constructed-directory :exists-p t)
                    (absolute-directory-from-path constructed-directory))))
             (if ensure-directory-exists-p
                 (absolute-directory-from-path source-path :exists-p t)
                 (absolute-directory-from-path source-path))))))

(defun construct-destination-path
    (&key source-path destination-directory destination-file-name
       destination-file-prefix destination-file-suffix
       (destination-file-extension NIL extension-provided-p) (separator ”-”)
       use-source-file-name)
  ”Return a fully constructed and valid destination PATH, given
components.

Scenarios:

DESTINATION-DIRECTORY, -FILE-NAME and -FILE-EXTENSION are provided,
and DESTINATION-DIRECTORY is an absolute path, then construct a destination
path concatenating them, else return an error. This is the default desired
behaviour.

Return: ’DESTINATION-DIRECTORY/DESTINATION-FILE-NAME.DESTINATION-FILE-EXTENSION’

SOURCE-PATH, DESTINATION-FILE-NAME and -FILE-EXTENSION are provided,
and SOURCE-PATH (or it’s parent directory) is an absolute path, construct
a destination path by concatenating them, else return an error.

Return: ’parent directory of SOURCE-PATH/DESTINATION-FILE-NAME.DESTINATION-FILE-EXTENSION’

If DESTINATION-DIRECTORY is additionally supplied, and it is a relative path, then
construct a destination path including this.

Return: ’parent directory of SOURCE-PATH/DESTINATION-DIRECTORY/
```

```
          DESTINATION-FILE-NAME.DESTINATION-FILE-EXTENSION'

In all cases, the DESTINATION-FILE-NAME and DESTINATION-FILE-EXTENSION are
expected.

Optionally, a file name prefix and/or suffix may be provided to the destination
name, separated by the SEPARATOR string. ”
  (when
      (and (or destination-file-name (file-pathname-p source-path))
           (or destination-file-extension extension-provided-p))
    (make-pathname :directory (pathname-directory
                               (probe-file
                                (ensure-directories-exist
                                 (construct-destination-directory
                                  :source-path source-path
                                  :destination-directory destination-directory))))
                   :name (construct-file-name
                          () ”-”
                          destination-file-prefix
                          (or
                           destination-file-name
                           (nth-value 2
                                      (split-path source-path)))
                          destination-file-suffix)
                   :type destination-file-extension)))

(defun construct-file-name (file-extension separator
                            &rest name-components)
  ”Construct file name given NAME-COMPONENTS, separated by SEPARATOR string,
returning combined name as a full name, including a FILE-EXTENSION string, if
it’s a non-empty string.

FILE-EXTENSION is mandatory, but may be empty (’’) or NIL. If it is provided, it
must be a string. No checking is done for illegal characters on your file
system, or if it’s a valid file extension.

SEPARATOR is mandatory, but may be empty (’’) or NIL. If it is provided, it must
be a string, and will be used to delineate individual NAME-COMPONENTS within the
destination file name.

NAME-COMPONENTS is not mandatory, but the function will immediately return NIL
if no components are provided. Components may be any number of: numbers, strings
or NIL; NIL values and zero-length strings will be pruned prior to the
construction of the file name.

CONSTRUCT-FILE-NAME returns a string with the combined file name, or NIL if any
conditions aren’t met.”
  (flet ((file-name (separator &rest components)
           (join
            (remove-if #’(lambda (component)
                           (or (null component)
                               (and (stringp component)
                                    (zerop (length component)))))
```

```
                      name-components)
           :separator separator)))
    (when name-components
      (if (and file-extension
               (not (zerop (length file-extension))))
          (nth-value 1
                     (split-path
                      (make-pathname
                       :name (file-name separator name-components)
                       :type file-extension)))
          (file-name separator name-components)))))
```

6 FFmpeg Interface

```
(defvar *ffmpeg-binary* ()
  "Store the absolute location of the ffmpeg binary for the duration of the session. If the binary d

#|
COMMAND EXECUTION

Build a call to a system binary, using the binary name
and any number of arguments passed in.
|#
(defun execute-command (program &rest arguments)
  "Execute binary specified by COMMAND, with any arguments passed
in as ARGUMENTS.

Return 3 values:
0: the result of the command's OUTPUT or NIL on non-zero exit status
1: the ERROR-OUTPUT string, or NIL
2: the actual EXIT-CODE of the process

Always directly execute the command, rather than calling a shell,
returning the entire output stream as a string stripped of any
newlines."
  (let ((command (or
                  (when arguments
                    (append (list program) arguments))
                  program)))
    (unless (zerop (length command))
      (multiple-value-bind (output error-output exit-status)
          (run-program command
                       :ignore-error-status t
                       :output '(:string :stripped t)
                       :error-output '(:string :stripped t)
                       :force-shell NIL)
        (values (or (and (zerop exit-status)
                         output)
                    NIL)
                error-output exit-status)))))

#|
Define a getter for the ffmpeg binary
|#

(defun find-ffmpeg ()
  ”Set *FFMPEG-BINARY* to the location of the program binary,
or leave as NIL if not found.”
  (unless *ffmpeg-binary*
    (setf *ffmpeg-binary* (execute-command ”which” ”ffmpeg”)))
  *ffmpeg-binary*)

#|
METADATA CONSTRUCTION

Prepare all aspects of video conversion:
- metadata
- global options
- local, per-format options
- per-file options
|#
(defvar *metadata-key-list*
    '(title date copyright artist album-artist author
      composer publisher album comment synopsis
      description content-type
      genre make model location grouping show
      episode-id episode-sort season lyrics language
      compilation network media-type hd-video
      gapless-playback)
  ”Define full list of accepted metatag KEYs.”)

(defmacro construct-metadata-plist-builder ()
  ”Construct metadata builder, from list of all accepted metatag
KEYs.

This macro defines a new function BUILD-METADATA-LIST, of the form:

(defun construct-metadata-list (&key title date ...)
  (mapcan #'(lambda (sublist) sublist)
    (list (when title (list :title title))
          (when date (list :date date))
          ...)))

which is a simple function prompting the user with all the legal values,
each of which, when supplied, is converted to a property list.
MAPCAN then takes each non-nil property list to create a broad, single-level
superlist.”
  `(defun construct-metadata-plist ,(append '(&key) *metadata-key-list*)
     (mapcan #'(lambda (sublist) sublist)
                (list
                 ,@(loop for i in *metadata-key-list*
                         collecting (list 'when i
                                          (list 'list
                                                (intern (string-upcase i) ”KEYWORD”)
                                                i)))))))
```

```
(defmacro construct-metadata-ffmpeg-args-builder ()
  `(defun construct-metadata-ffmpeg-args-list ,(append '(&key) *metadata-key-list*)
     (remove-if 'null
                (list
                 ,@(loop for i in *metadata-key-list*
                         collecting (list 'when i
                                          `(apply
                                            'concatenate
                                            (list 'string "-metadata "
                                                  ,(string->snake-case
                                                    (string-downcase
                                                     (symbol-name i)))
                                                  "="
                                                  (funcall 'string-escape ,i)))))))))

(construct-metadata-plist-builder)
(construct-metadata-ffmpeg-args-builder)

#|
FFMPEG PROFILES

Create a set of commonly used options, enabling quick generation of similarly-encoded
video files
|#
(defvar *global-ffmpeg-options*
  '("-y"                                ; Overwrite output without asking
    )
  "Set global ffmpeg options.")

(defvar *ffmpeg-web-video-sharing-hq-profile*
  '(:profile-name "Web video-sharing H264, high-quality"
    :profile-description "For use on video-sharing web-sites, H.264-encoded, full resolution, high q
    :file-name-profile-suffix "VHQ"
    :file-name-extension "mp4"
    :profile-options
    (;; Use the libx264 codec
     "-c:v libx264"

     ;; Set output to 8 bits per pixel, 4:2:0 chroma subsampling
     "-pix_fmt yuv420p"

     ;; An optional setting which limits the output to a specific H.264 profile.
     ;; Current profiles include: baseline, main, high, high10, high422, high444
     "-profile:v high"

     ;; A baseline profile with a level of 3.0 disables some advanced features to
     ;; provide better device compatibility
     "-level 3.0"

     ;; Constant rate factor in a scale of 0-51, where 0 is lossless, 51 worst possible,
     ;; and the default is 23
     "-crf 15"
```

```
      ;; A preset is a collection of options providing a certain speed to compression
      ;; ratio. A slower preset will achieve better quality and comression. Presets in
      ;; descending order are:  ultrafast,superfast, veryfast, faster, fast, medium,
      ;; slow, slower, veryslow, placebo. Medium is default
      "-preset veryslow"

      ;; Use the native (not linked) AAC encoder for audio
      "-c:a aac"

      ;; Convert the audio stream to a bitrate of 192k
      "-b:a 320k"

      ;; When the output video is going to be viewed on a browser, this output option
      ;; moves some header information to the beginning of the file, allowing the
      ;; browser to start playing the file even before it's completely downloaded
      "-movflags +faststart"))
   "Define an ffmpeg conversion profile for video-sharing web sites, using the H.264 encoder.")

 (defvar *ffmpeg-web-h264-profile*
   '(:profile-name "Web H264 1080p medium"
     :profile-description "For website display, H.264-encoded, 1080 wide, progressive medium quality"
     :file-name-profile-suffix "1080"
     :file-name-extension "mp4"
     :profile-options
     ("-vf scale=1080:-2"                ; Scale to 1080 pixels horizontally

      ;; Use the libx264 codec
      "-c:v libx264"

      ;; Set output to 8 bits per pixel, 4:2:0 chroma subsampling
      "-pix_fmt yuv420p"

      ;; An optional setting which limits the output to a specific H.264 profile.
      ;; Current profiles include: baseline, main, high, high10, high422, high444
      "-profile:v baseline"

      ;; A baseline profile with a level of 3.0 disables some advanced features to
      ;; provide better device compatibility
      "-level 3.0"

      ;; Constant rate factor in a scale of 0-51, where 0 is lossless, 51 worst possible,
      ;; and the default is 23
      "-crf 23"

      ;; A preset is a collection of options providing a certain speed to compression
      ;; ratio. A slower preset will achieve better quality and comression. Presets in
      ;; descending order are:  ultrafast,superfast, veryfast, faster, fast, medium,
      ;; slow, slower, veryslow, placebo. Medium is default
      "-preset veryslow"

      ;; Use the native (not linked) AAC encoder for audio
      "-c:a aac"
      ;; Convert the audio stream to a bitrate of 192k
      ”-b:a 192k”

      ;; When the output video is going to be viewed on a browser, this output option
      ;; moves some header information to the beginning of the file, allowing the
      ;; browser to start playing the file even before it’s completely downloaded
      ”-movflags +faststart”))
   ”Define an ffmpeg conversion profile for on web-site use, using the H.264 encoder.”)

 (defvar *ffmpeg-web-vp8-webm-profile*
   ’(:profile-name ”Web WebM VP8 1080p medium”
     :profile-description ”For website display, WebM-encoded (VP8), 1080 wide, progressive medium qua
     :file-name-profile-suffix ”1080”
     :file-name-extension ”webm”
     :profile-options
     (”-vf scale=1080:-2”                ; Scale to 1080 pixels horizontally

      ;; Use the libvpx codec
      ”-c:v libvpx”

      ;; Set output to 8 bits per pixel, 4:2:0 chroma subsampling
      ”-pix_fmt yuv420p”

      ;; To tweak the quality more finely, Q, or quantisation parameters can be
      ;; supplied in the form of -qmin and -qmax arguments, with a range of values
      ;; from 0-63, with 0 as best and 63 as worst
      ”-qmin 0 -qmax 25”

      ;; Constant rate factor in a scale of 4-63, where 4 is best, 63 worst possible
      ”-crf 4”

      ;; Supply an additional ’target’ variable bit rate for the encoder to try
      ;; and reach (in Mbit/s)
      ”-b:v 1M”

      ;; Use the libvorbis codec for audio
      ”-c:a libvorbis”

      ;; Set the audio quality level in a range of 0-10 where 10 is highest,
      ;; and 0 is lowest. 3-6 is a good range, the default is 3
      ”-q:a 7”))
   ”Define an ffmpeg conversion profile for on web-site use, using the VP8 WebM encoder.”)

 (defvar *ffmpeg-web-theora-vorbis-profile*
   ’(:profile-name ”Web Theora-Vorbis 1080p medium”
     :profile-description ”For website display, Theora/Vorbis-encoded, 1080 wide, progressive medium
     :file-name-profile-suffix ”1080”
     :file-name-extension ”ogv”
     :profile-options
     (”-vf scale=1080:-2”                ; Scale to 1080 pixels horizontally

      ;; Use the libtheora codec for video
      ”-c:v libtheora”

      ;; Set output to 8 bits per pixel, 4:2:0 chroma subsampling
      ”-pix_fmt yuv420p”

      ;; Video quality constant rate factor in a scale of 0-10, where 10 is highest
      ;; quality, 0 worst, with 5-7 a good range
      ”-q:v 6”

      ;; A preset is a collection of options providing a certain speed to compression
      ;; ratio. A slower preset will achieve better quality and comression. Presets in
      ;; descending order are:  ultrafast,superfast, veryfast, faster, fast, medium,
      ;; slow, slower, veryslow, placebo. Medium is default
      ”-preset veryslow”

      ;; Use the libvorbis codec for audio
      ”-c:a libvorbis”

      ;; Audio quality constant rate factor in a range of 0-10, where 10 is highest
      ;; quality, 0 worst, with 3-6 a good range
      ”-q:a 7”))
   ”Define an ffmpeg conversion profile for on web-site use, using the Theora/Vorbis
 encoders.”)

 (defvar *ffmpeg-web-profiles*
   ’(*ffmpeg-web-h264-profile* *ffmpeg-web-vp8-webm-profile*
     *ffmpeg-web-theora-vorbis-profile*)
   ”Define list of all profiles needed for a web export.”)

 #|
 STRING MANIPULATION

 A series of utility functions changing strings to different formats, as needed for
 command-line use, or for configuration files with different requirements of snake-,
 kebab-, camel-case or other string representation
 |#
 (defun string->snake-case (string &key case)
   ”Returns STRING in snake-case”
   (snake-case (join (split string ”-”) :separator ” ”)))

 (defun string-escape (string)
   ”Protect a string by escaping it in another layer of quotes.”
   (concatenate ’string ”\”” string ”\””))

 #|
 VIDEO ENCODING

 Functions which actually do the work of compiling a string of binary, arguments and
 other options, needed for the command-line
 |#
 (defun run-conversion (input-file
                        &key output-file-location output-file-name
                          output-file-prefix output-file-suffix
                          output-file-extension (separator "-")
                          profile metadata-list simulate-only)
   "Converts INPUT-FILE video to output."
   (when (and input-file (probe-file input-file))
     (let ((command-to-execute
             (join
              (append
               (list (find-ffmpeg))
               *global-ffmpeg-options*
               (list "-i" (string-escape
                           (namestring
                            (probe-file input-file))))
               (getf profile :profile-options)
               metadata-list
               (list
                (string-escape
                 (namestring
                  (construct-destination-path
                   :source-path input-file
                   :destination-directory output-file-location
                   :destination-file-name output-file-name
                   :destination-file-prefix output-file-prefix
                   :destination-file-suffix (or output-file-suffix
                                               (getf profile :file-name-profile-suffix))
                   :destination-file-extension (or output-file-extension
                                                   (getf profile :file-name-extension))
                   :separator separator)))))
              :separator " ")))
       (if simulate-only
           command-to-execute
           (execute-command command-to-execute)))))

 (defun run-batch-conversion (input-file
                              &key output-file-location output-file-name
                                output-file-prefix output-file-suffix
                                (separator "-") profile-list
                                metadata-list simulate-only)
   "Batch convert INPUT-FILE to each output profile specified in
 PROFILE-LIST"
   (mapcar #'(lambda (profile)
               (run-conversion input-file
                               :output-file-location output-file-location
                               :output-file-prefix output-file-prefix
                               :output-file-name output-file-name
                               :output-file-suffix output-file-suffix
                               :separator separator
                               :profile (eval profile)
                               :metadata-list metadata-list
                               :simulate-only simulate-only))
           profile-list))
```