» ExifTool Organise your photo collection in mere minutes with our handy scripts

ExifTool: Sort

Badly organised photo collections can induce enough stress to make you sick. Thankfully, **Marco Fioretti** has found the cure.

› This is what you get when you ask 'friends' and family to contribute to your photo collection: a mess!

Our expert

Marco Fioretti Is the author of *The Family Guide to Digital Freedom*. He's also a free software activist and programmer.

Not that long ago, photograph collections were made up of lots of bits of glossy paper with ink on them, which had been painstakingly amassed over years in cardboard boxes and professional-looking albums. Nowadays, the advent of affordable digital cameras means that we can produce hundreds or thousands of new photos every year. Despite being digital, all these need to be archived too.

Tools such as *Digikam*, while powerful, may not be the most efficient way to start filing all those pictures. Thus, in this tutorial, I'll explain a semi-automatic method to rename, sort and archive photographs to get you going. It's based on the *ExifTool* utility and two simple scripts, all of which are included on the disc. The only requirement to use them is very basic knowledge of shell scripting.

What are the scripts for?

My scripts do only a few things, but they're things that would consume most of your time if you tried to do them manually. Firstly, they'll rename and tag all your picture files by date. Then they'll automatically save each picture in a set of folders that are also arranged by date in a year, month, day hierarchy. If you want to take this further, the scripts can be use to add comments, ratings and tags to your photographs. GUI tools such *Digikam* are often better for that kind of work, but you can save a huge amount of time if you start using them after you've pre-indexed your collection.

In and of themselves, the scripts are quite simple to use. As long, that is, as you know the rationale behind them and a thing or two about photographic metadata. In fact, the most complicated part of this tutorial isn't understanding how the scripts work, but why they have to do certain things.

Starting with chaos

OK, let's say that one day you wake up and decide to make one, neatly ordered digital collection of all the paper and digital photographs you care about. By hand, that isn't just a long adventure, it's also a desperate one. The chances are that you will face any combination of the following tasks: the first is how to digitise your physical pictures, which is boring and time consuming, but conceptually simple. To speed things up, you can scan four photographs at a time with any A4 scanner and save the results with unique names using the method I explained in **LXF123**.

Another common need is to make sense of all the digital pictures sent to you by friends with different digital cameras or even the same model as yours. The former will fill your drive with files bearing every conceivable, and thus inconsistent, naming scheme known to man. Hundreds of files will be named something like **XYZ_001.JPG** to **XYZ_153.JPG**, while others will be **P00001.jpeg** to **P99999.jpeg** and so on. If you're unlucky enough to own the same camera as your friends, you risk getting a bunch of files with the same names as yours, inviting disastrous overwriting situations unless you're really careful.

The worst case scenario combines the most annoying characteristics of the two problems mentioned above. This is when you receive files of scanned paper pictures in every graphic format conceived since the inception of the transistor. What's more, their names will be light years away from anything useful or consistent. I can only imagine that whoever sent these files to you hates you subconsciously.

your pictures

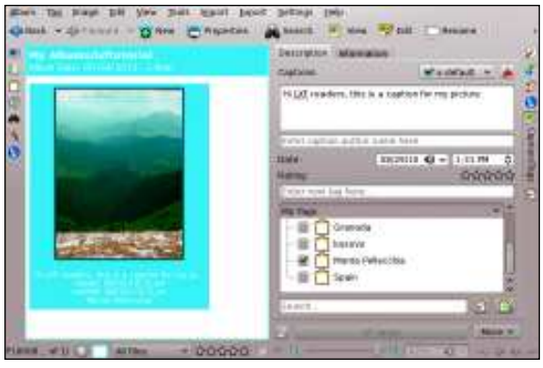
Renaming this mess is our first task, then. After all, consistent file naming is the basis for efficient and portable picture management. We have a few criteria for this: foremost is that you want your pictures to remain individually available as files in standard formats. Burying photos in the depths of a database may give better performance in the short term, but it's a no-no if you care about making backing them up simple and future proofing your collection.

For similar reasons, and since files need unique names, we should avoid characters that aren't alphanumeric in our filenames. Descriptive titles are out too – while names such as **Mom.jpg** and **Mom_graduation.jpg** might seem to make sense today, wait just a few years and you'll start wondering, 'Whose mom? Mine or my spouse's?'

How about progressive numbers, then? No. They'll stop being progressive, and thus meaningful, the first time you'll have to add some picture to an already existing album. Nope, the most portable, scalable and future-proof naming scheme that we know of is simply to give each file a name corresponding to the exact moment at which the picture was taken. Filenames in the format YYYYMMDDHHMMSS are portable; not too long; always sort themselves by date in all programs, even if you remove or add pictures to a gallery; and are perfect to make backups from that are sorted chronologically. Putting all the pictures taken on the same day or in the same month in one folder with a name such as **Year/Month** or **YearMonth** only makes indexing easier.

With our naming convention established, let's talk metadata, that is extra information about each picture. Metadata can be almost anything, from author and copyright declarations to camera settings or category labels. There are three main standards for picture metadata, but here we'll focus on a common scenario: pictures in JPEG format with EXIF (Exchangeable Image File Format) metadata.

In general, photograph metadata can either be kept within a file (as happens with EXIF) or in a database. The first solution makes sure that metadata isn't lost if you move your photographs. The second makes searches and other operations much faster. Personally, I use both systems simultaneously. I write metadata in EXIF format inside the files first, then, when I open my galleries with *Digikam*, I tell it to copy all that stuff into its own internal database.



› ***Digikam*** **has a powerful tagging and metadata interface, but is the mouse really the only way?**

What about spaces?

The main script explained here will fail if a filename contains spaces or weird characters. I've deliberately avoided this case in the script because, while both Linux and the shell can handle such characters, avoiding them as a general rule makes it much easier and faster to write all kinds of cool, advanced scripts. Therefore, I never generate weird file names on my computer, and rename all files I get from others as soon as I can. You don't need to be as extreme yourself, though. To manage files with weird names, just replace the **for PIC...** line in my script with the first four lines of the one you'll find at: **www.cyberciti.biz/tips/handling-filenames-with-spaces-in-bash.html**.

Using the scripts

Now let's deal with a specific example: the situation you can see in the screenshot on the previous page. Here, we have a bunch of images in different formats, all of which are waiting for order, purpose and identity from you. Note that our start folder only contains four photo files (plus a text one, whose purpose will become clear in a moment) for clarity, but the script would work even if there were thousands. Some of the images aren't even photographs, just random screenshots. I've done that on purpose, to prove a point. If you need to, you can transform any picture file into something much more usable: a file that any photo manager will handle – metadata and all – just like a file produced by a digital camera.

Here's how the script works: to begin with, it puts all the pictures that need sorting in a directory created just for this task. Then, and this is the only manual work you need to do, you must prepare a plain text file called **notes_file.txt**.

This file is necessary because we're renaming, sorting and archiving photographs according to the moment at which they were created. For digital photographs, that data is written by the camera in an EXIF tag called DateTimeOriginal. In all other cases, it up to us to provide an approximate value. These are exactly what you need to write into **notes_file.txt**:

```
$ cat notes_file.txt
joe.jpg  | 2001:01:01
Mom.JPEG | 1998:12:01
Mike.png
Dad.tiff | 2005:03:28
```

As you can see, you just have to write the names of all the picture files that are in the folder, one per line. You may get that listing with command like **ls -l | cut -c45- > notes_file.txt**. Then, after a pipe character, add the time when that picture was taken in the YYYY:MM:DD format to each line. If

there are two or more files with the same date, write them on consecutive lines and only add the date to the first on in the list. Once everything's ready, launch the script at a command prompt, passing the complete path to that directory as the first and only argument:

```
$ photo_archiver.sh $HOME/lxf_demo
```

The script will work on each graphic file as shown in the flowchart on the right.

Breaking the script down

The first thing that happens is preservation of the original file: we make a copy to work on. Then, if necessary, that copy is converted into JPEG format, because we need files that can store EXIF tags. This step is done by the **convert** utility, which is part of the *ImageMagick* suite (**www.imagemagick.org**):

```
convert /tmp/my_tmp_photo  /tmp/my_tmp_pic.jpg
```

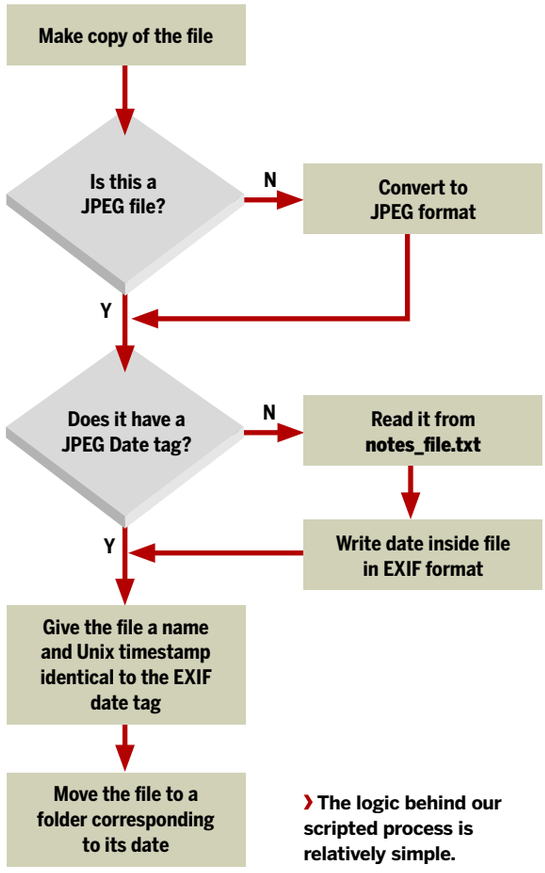
Next, we use *ExifTool* to manage EXIF tags:

```
1 DATE=`exiftool -s -f -DateTimeOriginal /tmp/my_tmp_pic.
jpg | cut -d: -f2-`
2
3 if [ "$DATE" == " -" ]
4          DATE_FROM_NOTES=`$RETRIEVE_DATE notes_
file.txt $PIC`
5     if [ "$DATE_FROM_NOTES" == "NO_DATE_FOUND" ]
6          then
7              echo "NO_DATE_FOUND for $PIC, sorry!"
8              exit
9          else
10             exiftool -DateTimeOriginal="$DATE_FROM_
NOTES 12:00:00" /tmp/my_tmp_pic.jpg
11    fi
12 fi
```

The first line above tells *ExifTool* to extract the EXIF parameter called **DateTimeOriginal** from the JPEG file and store it, after removing all spaces and unnecessary parts, inside the **$DATE** variable. If this operation returns an empty date, we ask the **retrieve_date_from_file.pl** script (included on the DVD) to extract the approximate date from **notes_file.txt**. Then we run *ExifTool* again to write that value inside the JPEG file. Please note that the script always uses the same hour (**12:00:00**).If you want more accuracy, enter more precise notes in **notes_file.txt** and modify the second *ExifTool* invocation to use them.

Now we have a file, called **my_tmp_pic.jpg**, that is in JPEG format and contains a shot date in the DateTimeOriginal EXIF variable. Then the next part of the script is applied:

```
1 NEWDATE=`exiftool -s -f -DateTimeOriginal /tmp/my_
tmp_pic.jpg | cut -d: -f2- |  tr -d " :"`
2 PARTIAL_DATE=`echo $NEWDATE | cut -c1-12`
```



The logic behind our scripted process is relatively simple.

```
3 SECNDS=`echo "$NEWDATE" | cut -c13-14`
4 mv /tmp/my_tmp_pic.jpg /tmp/$NEWDATE.jpg
5 touch -t "$PARTIAL_DATE.$SECNDS" /tmp/$NEWDATE.
jpg
```

First, we read the date again, remove all spaces and colons (note the **tr -d** command at the end of the first line) and call the result **$NEWDATE**. The format of this variable is YYYYMMDDHHMMSS. The first 12 characters of this string, which describe the date with one-minute accuracy, go into **$PARTIAL_DATE**. The last two, the seconds, go into **$SECNDS**. You'll understand this in a moment.

The reason to generate **$NEWDATE** is evident in line 4 of the short listing above: to rename the temporary file as **$NEWDATE.jpg**. Line 5 does the last bit of not-so-black magic: it changes the timestamp of that file to the same value, using the **touch** command. Unfortunately, this utility recognises seconds only after a period. This is why we are forced to generate **$PARTIAL_DATE** and **$SECNDS** separately. To summarise what happens, after running the script **Mom.JPEG** becomes **19981201120001.jpg**:

```
$ ls -l 19981201120001.jpg
-rw-rw-r-- 1 marco marco 61909 Dec  1  1998
19981201120001.jpg
$ exiftool -s -f -DateTimeOriginal 19981201120001.jpg | cut
-d: -f2- |  tr -d " :"
19981201120000
```

See what I mean? Now we have a file whose name, timestamp and original date in EXIF format all match! The last thing the **photo_archiver.sh** script does is to put each

How do you remove EXIF data?

Writing comments, tags and other information inside your photographs is, technically speaking, a very sound strategy. However, from a PR point of view, it may also be a suicidal one. The reason being that EXIF is a worldwide open standard and EXIF metadata isn't encrypted. This means that whoever gets a copy of your picture can also read all its metadata. Remembering this fact of life will keep you happy. You're still entitled to tag your uncle's portrait as 'big loser', but before sending him that picture by email or uploading it to Flickr, make a copy and then run the following command, which will erase the content of all its EXIF field:

```
exiftool -all= uncle_portrait.jpg
```

picture in a folder with a proper name, creating it if it didn't already exist:

```
1 YEAR=`echo $NEWDATE | cut -c1-4`
2 MONTH=`echo $NEWDATE | cut -c5-6`
3 DAY=`echo $NEWDATE | cut -c7-8`
4
5 if [ ! -d "$BASE_PHOTO_DIR/$YEAR/$MONTH/$DAY" ]
6  then
7  mkdir -p $BASE_PHOTO_DIR/$YEAR/$MONTH/$DAY
8 fi
9 mv /tmp/$NEWDATE.jpg $BASE_PHOTO_
DIR/$YEAR/$MONTH/$DAY
```

The first three lines here split **$NEWDATE** in its year, month and day parts. Lines 5 to 7 create the directory if needed, according to the file date, and move it there. In my case, **19981201120001.jpg** goes inside of **$BASE_PHOTO_DIR/1998/12/01**.

Now let's stand back and admire the results in the screenshot below. The left-hand pane shows that the script created the right hierarchy of albums inside **$BASE_PHOTO_DIR (lxf_demo)**, according to preexisting EXIF tags or the contents of **notes_file.txt**. In the central pane, and the bottom-left corner, you can see that filenames and timestamps were modified accordingly. Finally, the EXIF panel on the right shows that *Digikam* recognises and will display the original date that the script wrote as EXIF data.

What else should you know?

Attentive readers will have noticed an apparent bug in the code shown so far. What if there's more than one picture with the same value of DateTimeOriginal? Won't each of them overwrite the previous one if they all get the same **YYYYMMDDHHMMSS.jpg** name? Despair not, take a look at the screenshot below of the process's final result and you'll notice that there are two files, not just one. That is because the script does check if a file with a certain name already exists and, when needed, it does a dirty but effective trick. Since the filename is an integer, if **N.jpg** exists, the script saves the current photograph to **N+1.jpg**.

Of course, we still live in a universe where valid second values only go from 0 to 59. What if there are more than 60 pictures with exactly the same value of DateTimeOriginal? Easy, you end up with all those pictures having the same, valid Unix timestamp (like **19981201120001.jpg**), but only the first 60 will have a filename that's also a valid timestamp. If this is a real problem and not just a theoretical one, it is easy enough to modify the code to have, for example, unique names thanks to extra suffixes such as **19981201120001.aa.jpg**, **19981201120001.ab.jpg**...

Finally, JPEG is a lossy format. If you have high quality photographs, making JPEG versions is extremely convenient, but you also need to keep each original safe and linked to its JPEG copy. The way to do this – which is left as an exercise for you, since there isn't the space to cover it properly here – is to rename the raw file with the same YYYYMMDDHHMMSS string, but keep the original extension.

How to generalise this method

The script I've just explained is made possible by one single thing: *ExifTool*'s ability to read and write date tags. EXIF metadata, however, can be used to store much more than just dates. Since *ExifTool* supports all EXIF parameters, you can use it to automatically set any others you desire. For example you could add a description to a photograph from the command prompt in this way:

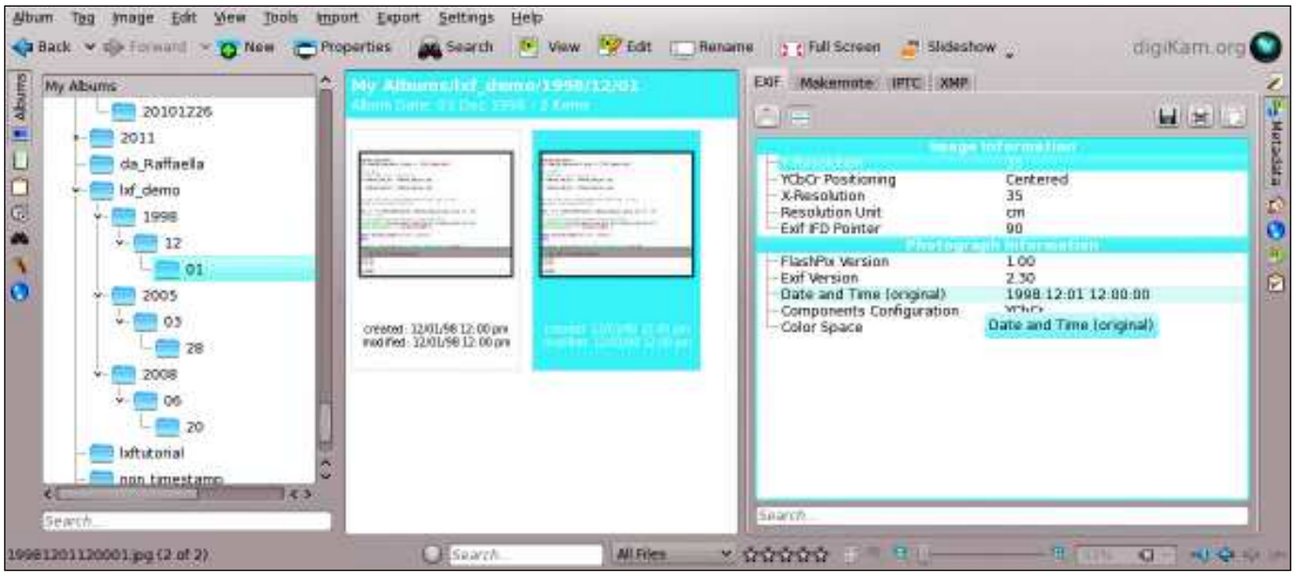
```
$ exiftool '-Description=this, instead, is a caption inserted
with exiftool' P1090097.JPG
$ exiftool -Keywords='Montefeltro, family' P1090097.JPG
```

The flexibility of *ExifTool* doesn't end there. I automatically added latitude and longitude data to my photographs using the script from my geotagging tutorial (in **LXF112**). Now any software capable of geolocalisation can show where my pictures were taken on a map. Isn't *ExifTool* wonderful? LXF

Stuck?

If you've run the script and things aren't quite as you expected, don't despair. First off, *Digikam* – or any other photo manager with its own database – won't necessarily sync the content of EXIF tags straight away. For *Digikam*, select Images > Reread Metadata From Images in the main menu to see your changes.

You should also know that both EXIF and *Digikam* have more than one date tag. For example, there's one tag to store the date when a file was last modified. Look at the EXIF panel in *Digikam* or in the *ExifTool* documentation to find out more. Personally, I have never needed to mess directly with those extra tags. If you want to process them, however, all you need is to add a few extra lines to the script.



› The end result: all your files get decent names, tags and places. Life is good!