iPhone-exif Developer Guide
v 0.1

Table of Contents

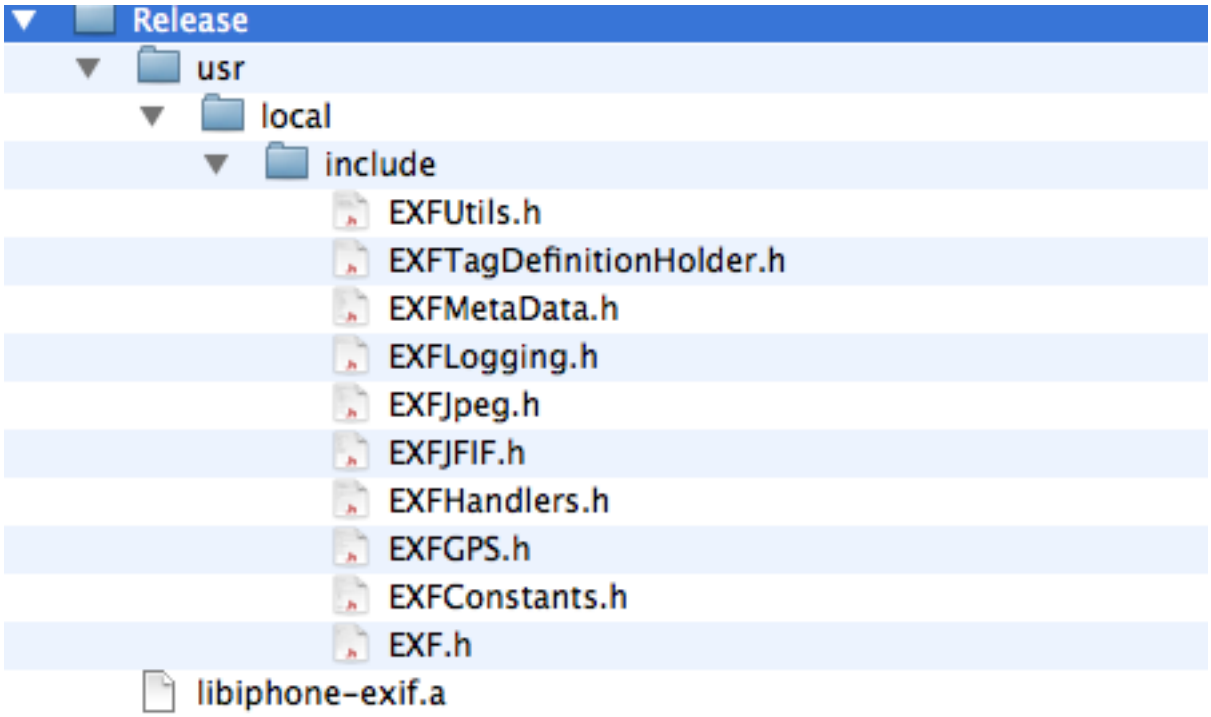
Using the Library in your Project

The library can be used in two ways:

1. Import the binary as a static library
2. Embed the source directly

Use as a static library

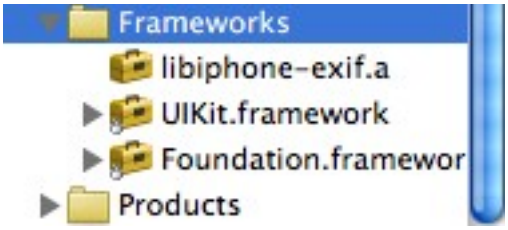
To use as a static library in your project you must download the zip file and unzip to a suitable location. This should give you the following layout:



The libiphone-exif.a is the static library, the include directory contains the header files.

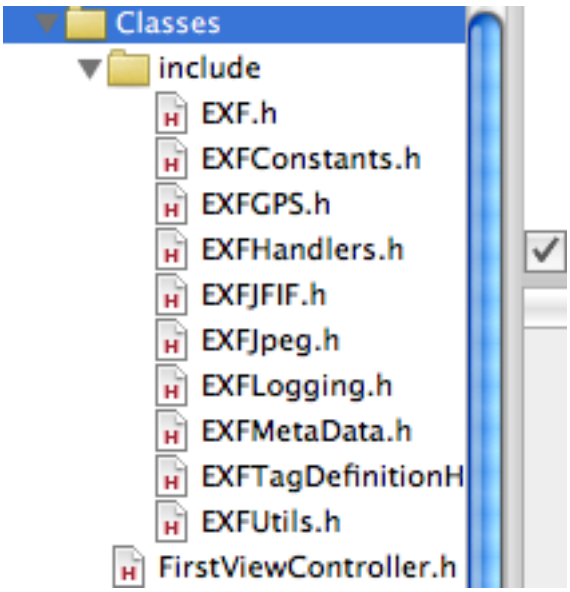
To use this in Xcode first add the library to the frameworks library.

1. Select the frameworks folder.
2. Select add > existing file
3. Make sure the copy file option is selected in the dialog box.
4. You should see the library appear in the frameworks folder as a toolbox icon. e.g



We now need to include the header files in our project.

1. Select the Classes directory.
2. Click add existing files
3. Navigate to the unzip folder and select the include directory containing the header files.
4. Ensure the copy option is selected.
5. You should see the include directory appear under the Classes directory. (If you have a personal preference you can place these wherever you want.



Use as source code

First you need to check out the source code from SVN. Follow the instructions on the google code page to check out the code.

The Classes directory contains all the header and implementation files. Simply drag these files into a suitable Group folder in Xcode.

These files will then be built directly into your project.

API Guide

The API is defined to be as Objective-C friendly as possible. The internals of the library are a mix of C and objective , but as a user of the library it is easier if you stick to the Objective-C APIs.

The classes we are interested in are most probably:

- EXFJpeg.h
- EXFMetaData.h
- EXFGPS.h
- EXFConstants.h

If you are defining your own handlers (detailed later) you will also need to look into

- EXFHandlers.h

In general the best way to use the library is to import the EXF.h header file, which contains a link to all the other headers you will need.

So at the top of your files use the following directive

```
#import "EXF.h"
```

Parsing the EXIF Data

The EXFJpeg class is the root object in the API and is used to scan a Jpeg image to extract the EXIF, to provide a handle to the EXFMetaData object which allows us to manipulate the image data and to produce the new image data after we have completed all our manipulations.

The following code demonstrates how we scan the jpeg file for the EXIF information.

```
NSData * uiJpeg =  UIImageJPEGRepresentation (anImage, 1.0 );
EXFJpeg* jpegScanner = [[EXFJpeg alloc] init];
[jpegScanner scanImageData: anImage];
self.imageData = jpegScanner;
[jpegScanner release];
```

Here we extract the NSData representation of a UIImage, although we could originate the NSData directly from file without going via the UIImage representation first (currently, the iPhone UIImage will strip out existing EXIF information and resize the image to the view – DO NOT use UIImage if you want to save the amended data back to the file system, instead read the data directly from file).

Next we instantiate the EXFJpeg object and pass the NSData object to the instance. This method extracts the EXIF data in the image and constructs the EXFMetaData object that allows you to manipulate the tags.

The Exif MetaData

The Exif data is predicated on the idea of a Tag. The Exif data is a block of bytes representing the set of tags, with each tag having an numeric ID and a data value. Each data value can be of a set data type and a set number of bytes. In order to manipulate the tag we must therefore know what the data type is and what its ID is.

The Ids and types are detailed in the EXIF specification. However, the library also provides some help in this matter. You do not need to be familiar with the whole specification, but you at least need to know the numeric ID for the tags you wish to manipulate and the data types that are allowed.

The library will reject invalid data types for a tag.

The EXIF Tag definition

The EXFMetaData object provides us with a way of finding out the specification structure for each Tag. We use the tagDefinition method to give us a representation. e.g:

```
EXFTag* tag = [self.imageData.exifMetaData tagDefinition:tagNum];

The tag object is defined in EXFConstants.h and is defined as:


@interface EXFTag : NSObject {


    EXFTagId tagId;
    EXFDataType dataType;
    int parentTagId;
    NSString* name;
    BOOL editable;
    int components;

}
```

The tag object can then be used to find the dataType, the short name and if appropriate a parent tag (we shall address the tag hierachy later in this document). The size of the data in each tag is specified as a combination of the number of bytes occupied by the data type * the number of components.

The possible data types are also defined in EXFConstants.h and are:

```
enum EXFDataType {
    FMT_BYTE =       1,
    FMT_STRING  =    2,
    FMT_USHORT  =    3,
    FMT_ULONG   =    4,
    FMT_URATIONAL  = 5,
    FMT_SBYTE      = 6,
    FMT_UNDEFINED  = 7,
    FMT_SSHORT     = 8,
    FMT_SLONG      = 9,
    FMT_SRATIONAL  =10,
    FMT_SINGLE     =11,
    FMT_DOUBLE     =12
};
```

In most cases the types are as sized as one would expect for the numeric types (except for the rational types which we will address later). And indeed as a programmer we only need to know that all integer types are represented in the library as NSNumbers. (Note the size of the NSNumber is restricted for the type and follows the usual type promotion rules – you cannot put a long value into a tag specified as a short, but a long can contain a value that is originated as a short.

String types are always ASCII characters and the library will treat all Strings as ASCII. The component aspect of the tag tells us if it is a single instance of the data type or an array. The number of components is always the exact number of entries in the array.

Therefore for most tags we can arrive at a few simple rules.

1. If the tag is an integer and has a component count of 1 it is a single NSNumber
2. If the tag is an integer and has a component count > 1 then it is an array of NSNumbers
3. The NSNumber must not overflow the numeric limit of the type
4. If it is a String it can only contain ASCII characters and the String length is the number specified by the component field.

Rational types are not stored in the exif data as floating point numbers as one might expect, instead they are represented as an array of fractions, with both the numerator and denominator specified as a long. In most cases we do not need to deal with this except for GPS values which will be detailed in its own section.

The unknown data type must either be handled by a specific regsitered handler, or it is treated as an opaque type and is provided as an NSData instance.

Although this seems complicated in practice we really only deal with NStrings, NSNumber and the custom GPS classes provided by the library. Any more custom behaviour will require a more detailed appreciation of the underlying byte details.

Reading Tag Data

The existing tag values can be retrieved using the EXFMetaData class in the following manner:

```
id value = [ self.imageData.exifMetaData tagValue: aTagId];
```

the format for the value, as we previosly showed can be a number of types. Normally we will know the type, but if we are testing an unknown tag and say want to display the value in a string we could do something like:

```
if ([value isKindOfClass:[NSArray class]]){
     // formater for array
     textValue =[NSString stringWithFormat:@"[%@]",[value
                componentsJoinedByString:@","]];
}else if ([value isKindOfClass:[NSData class]]){
     textValue =@"Binary";
}else{

     textValue = [NSString stringWithFormat:@"%@",value];
}
```

It is important to bear in mind that we may need to convert some numeric values into a more friendly text format (for example image orientation is numeric but displaying it as such is not that meaningful). We may then need to map some values to a text representation. In most cases this is quite straight forward as the possible values is usually predefined. So we could perhaps concatenate the tagId with a know value to act as a key to the desired text format e.g. imageOrientation (tag id == 274) has eight possible values in the Exif spec:

```
    [valueLookups setObject:@"Top Left" forKey:@"274.1"];
    [valueLookups setObject:@"Top Right"  forKey:@"274.2"];
    [valueLookups setObject:@"Bottom Right" forKey:@"274.3"];
    [valueLookups setObject:@"Bottom Left" forKey: @"274.4"];
    [valueLookups setObject:@"Left Top"  forKey:@"274.5"];
    [valueLookups setObject:@"Right Top" forKey:@"274.6"];
    [valueLookups setObject:@"Right Bottom" forKey:@"274.7" ];
    [valueLookups setObject:@"Left Bottom" forKey:@"274.8"  ];
```

Although the EXIF data has some hierarchy inherent in its structure, the library provides the interface of a flat tag space so tags in sub-tags will be returned with this call.

Writing Exif Data

Writing EXIF data is reasonably straight forward for most tags.

```
      [self.imageData.exifMetaData addTagValue: @"Apple" forKey:[NSNumber
                                    numberWithInt:EXIF_Make]];
```

In the above example we are writing a String value into the EXIF_MAKE tag. The Tag Ids are specified in the Constants.h file and are of the format EXIF_XXXX.

Numeric values are similarly simple:

```
[self.imageData.exifMetaData addTagValue: [NSNumber numberWithInt:1]
                    forKey:[NSNumber numberWithInt:EXIF_GPSAltitudeRef] ];
```

For tags that have component count > 1 that are numeric we must pass the data in an NSArray format. e.g:

```
NSMutableArray* array = [[NSMutableArray alloc] init];
[array addObject:[NSNumber numberWithChar:2] ];
[array addObject:[NSNumber numberWithChar:2] ];
[array addObject:[NSNumber numberWithChar:0] ];
[array addObject:[NSNumber numberWithChar:0] ];

[ self.imageData.exifMetaData addTagValue: array forKey:[NSNumber
                                    numberWithInt:EXIF_GPSVersion] ];

[array release];
```