iPhone-exif Developer Guide
v 0.8

Author: Steve Woodcock
Date: 13/07/2008

Table of Contents

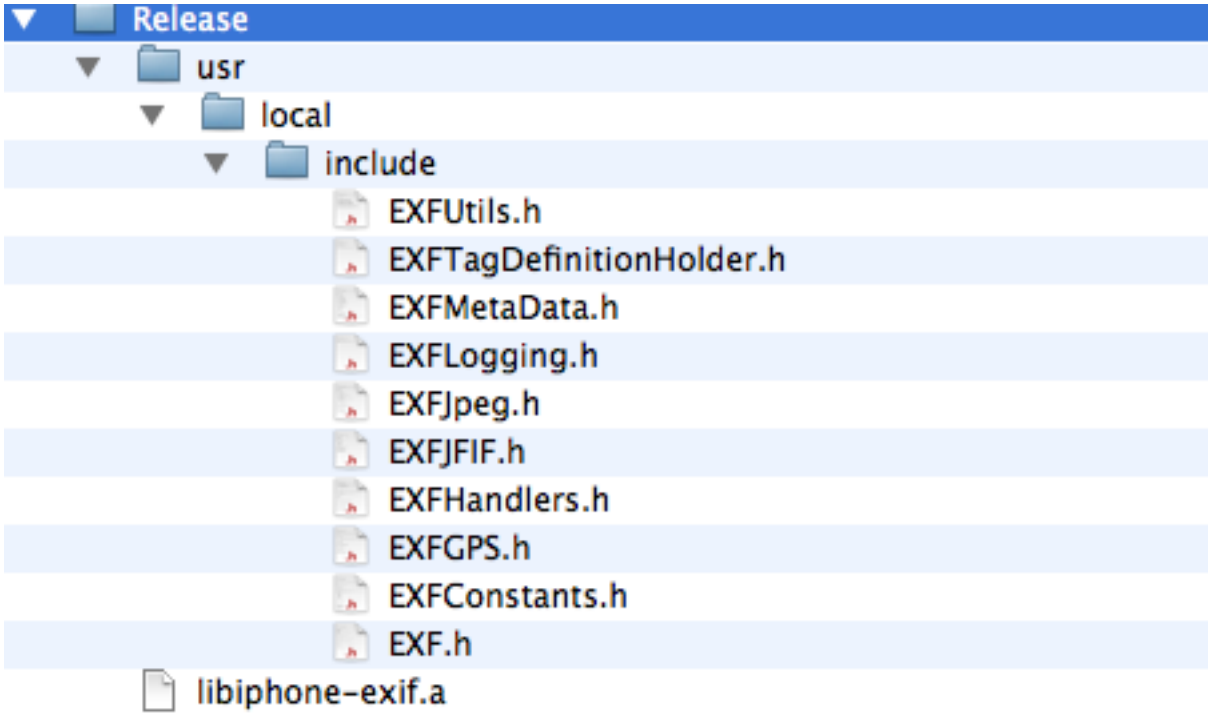
Using the Library in your Project

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

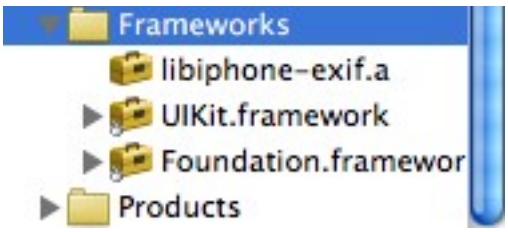
To use as a static library in your project you must download the zip file and unzip to a suitable location. This should give you the following layout:



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

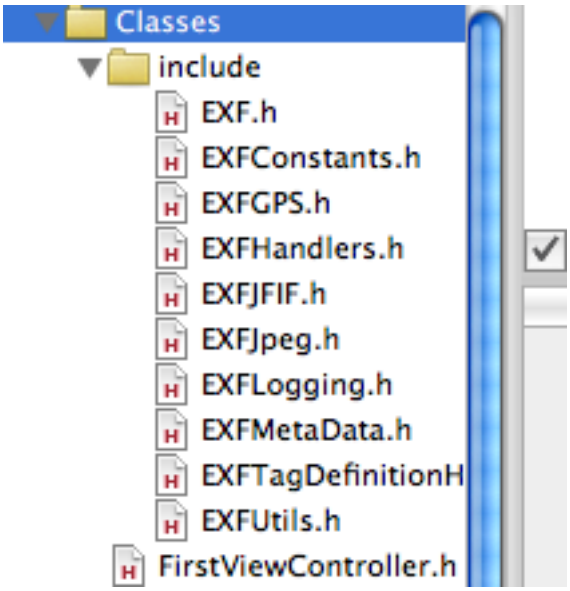
To use this in Xcode first add the library to the frameworks library.

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 most are 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"
```

Debug directive

Currently, the library requires you to specify a debug directive, which can be used to control the debug messages inside the library. This must be set using (set this to TRUE to enable messages):

```
BOOL gLogging = FALSE;
```

in the appDelegate.m class.

Parsing the EXIF Data from an Image

The EXFJpeg class is the root object in the API and is used to scan a Jpeg image to extract the EXIF tag data. This then gives us 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 as an NSData object).

Next we instantiate the EXFJpeg object and pass the NSData reference 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 stored as a block of bytes in the header portion of the JPEG Image file. This block of bytes contains the set of tags, with each tag consisting of a numeric ID and a data value. Each data value must be of a predefined data type and is a set number of bytes in length. 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 most awkward types to deal with are the rational number types. We will address this in the section that deals with the GPS tags. It is also possible to override any tag handling in the library using a custom handler, or manage your own tags or something non-standard like the maker's block inserted by a Camera.

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 Id is the numeric id of the tag as defined in the specification. The dataType is the expected type of the tag. The parentTagId defines the tag's position in the hierarchy (which will be addressed later in this guide). The short name of the tag is the name defined in the specification, the editable

flag allows the tag to be set as user editable and components defines the number of elements of each datatype in the tag. 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.

For instance, the long type is 4 bytes, and a component count of 3 would give us a total data type width of 12 bytes. Component counts of more than one usually represent an array (there are some exceptions which we will address later in the document).

The possible data types are 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 NSStrings 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 of thumb.

1. If the tag is an integer type and has a component count of 1 it is a single NSNumber
2. If the tag is an integer type and has a component count > 1 then it is an array of NSNumbers
3. The NSNumber content must not overflow the numeric limit of the type
4. If it is an NSString 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 always encoded and decoded as an NSData instance by the library.

Although this seems complicated, in practice we really mostly deal with NStrings, NSNumber and the custom GPS classes provided by the library. More complicated behaviour or custom handling will require a more complete understanding of the EXIF data and the underlying byte formats (see the section on custom tag handling).

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 previously showed can be one of 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 all 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"  ];
```

It is important to realise that although the EXIF data has some hierarchy inherent in its structure, the library provides the lookup of the tag vlaue as shown above as a flat tag space so tags that are really sub-tags in the hierarchy will be returned with this tagValue call without having to manually navigate the tag structure ourselves.

Writing Tag Data

Writing EXIF tag 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];
```

Note: String values are never arrays, each component is actually a character in the String. This covers most of the standard tags. The next section deals with the slightly more complicated requirements needed for the GPS types.

Dealing with GPS Data

The GPS Tags in the EXIF spec are defined as:

```
GPSLatitudeRef
GPSLatitude
GPSLongitudeRef
GPSLongitude
GPSAltitudeRef
GPSAltitude
GPSTimeStamp
GPSSatellites
GPSStatus
GPSMeasureMode
GPSDOP
GPSSpeedRef
GPSSpeed
```

```
GPSTrackRef
GPSTrack
GPSImgDirectionRef
GPSImgDirection
GPSMapDatum
GPSDestLatitudeRef
GPSDestLatitude
GPSDestLongitudeRef
GPSDestLongitude
GPSDestBearingRef
GPSDestBearing
GPSDestDistanceRef
GPSDestDistance
```

The actual location data is specified as two coordinates, one longitude and one latitude. This gives a point on the surface of the earth. Normally, longitude is 180W to 0 at the meridian to 180E degrees and latitude 90 at N Pole to 0 at the equator to 90 at the south pole. Sometimes this is represented as positive and negative numbers to represent which side of the equator or meridian the measurement is on. Indeed this is the format taken by the iphone. Additionally in the iPhone the latitude or longitude is a single double format number, so for example, a longitude will be a single double number e.g. -122.09345 .

However, The EXIF specification does not allow negative numbers and each point is represented as a pair of tags with the xxxRef tag representing a modifier for the corresponding value tag (E/W or N/S). For instance the GPSLongitudeRef and GPSLongitude tag together represent the longitude value.

Additionally, The format for the EXIF GPS is three rational numbers one each for degrees, minutes and seconds, not a single double number. While this sounds reasonable the spec goes on to say that each rational number is actually stored as a fraction comprising two longs rather than a floating point number. So, the iPhone representation of the GPS location as a single decimal entails conversion between these formats.

The library models the EXIF GPS format as a GPS location class, which contains three member variables of a Fraction type. The reason we do not store it as three doubles is that this will result in a loss of precision when round-tripping to and from the EXIF tag, so even though we can display it in this format for users, the Library tries to retain the original fraction representation in the tag.

The interface definition for this is:

```
@interface EXFGPSLoc : NSObject {
	EXFraction* degrees;
	EXFraction* minutes;
	EXFfraction* seconds
}
```

While this is fine in principle it can be a little awkward in practice. In order to set the GPS data the main the programming task is to construct an instance of a GPSLoc class and use this as the data type to pass into the EXIF Library e.g.

```
// Helper methods for location conversion
-(NSMutableArray*) createLocArray:(double) val{
      val = fabs(val);
      NSMutableArray* array = [[NSMutableArray alloc] init];
      double deg = (int)val;
      [array addObject:[NSNumber numberWithDouble:deg]];
      val = val - deg;
      val = val*60;
      double minutes = (int) val;
      [array addObject:[NSNumber numberWithDouble:minutes]];
      val = val - minutes;
      val = val *60;
      double seconds = val;
      [array addObject:[NSNumber numberWithDouble:seconds]];
      return array;
}

-(void) populateGPS: (EXFGPSLoc*)gpsLoc :(NSArray*) locArray{
      long numDenumArray[2];
      long* arrPtr = numDenumArray;
      [EXFUtils convertRationalToFraction:&arrPtr :[locArray objectAtIndex:0]];
      EXFraction* fract = [[EXFraction alloc] initWith:numDenumArray[0]
            :numDenumArray[1]];
      gpsLoc.degrees = fract;
      [fract release];
      [EXFUtils convertRationalToFraction:&arrPtr :[locArray objectAtIndex:1]];
      fract = [[EXFraction alloc] initWith:numDenumArray[0] :numDenumArray[1]];
      gpsLoc.minutes = fract;
      [fract release];
      [EXFUtils convertRationalToFraction:&arrPtr :[locArray objectAtIndex:2]];
      fract = [[EXFraction alloc] initWith:numDenumArray[0] :numDenumArray[1]];
      gpsLoc.seconds = fract;
      [fract release]
}
// end of helper methods

// adding GPS data to the Exif object
NSMutableArray* locArray = [self createLocArray:location.coordinate.latitude];
EXFGPSLoc* gpsLoc = [[EXFGPSLoc alloc] init];
[self populateGPS: gpsLoc :locArray];
[exifMetaData addTagValue:gpsLoc forKey:[NSNumber
numberWithInt:EXIF_GPSLatitude] ];
[gpsLoc release];
[locArray release];
```

In the above code we are converting the iPhone representation of the location.coordinate into an array representation of degrees, minutes and seconds, then populating the GPSLoc object and setting this into the exifMetaData object.

The createLocArray method first converts the decimal representation of the latitude from the iPhone, which looks something like -122.030731, to an array of NSNumbers. These are then passed to the populateGPS method which converts each NSNumber to a fraction and sets this into the GPSLoc instance. This is a simple process but a little tedious. Note: the conversion of each of these numbers to a fraction in the EXFUtils class uses a standard Euclidean GCD method.

The ref part of the tag pair is dealt with as a String e.g

```
location.coordinate.latitude;
if (location.coordinate.latitude <0.0){
      ref = @"S";
}else{
      ref =@"N";
}
[exifMetaData addTagValue: ref forKey:[NSNumber
numberWithInt:EXIF_GPSLatitudeRef] ];
```

Of course, this means that you have to have some knowledge of the Exif GPS tags, but the library shields you from the nitty-gritty of the underlying byte formats. Generally this is not too much of a problem, once you get the conversion routines written.

Custom Tag Handlers

While the processing of the tags in the library are sufficient for normal processing of numeric and String tags, there are times when we either want to override the behaviour of the library for a specific tag or set of tags or have the ability add our own tags to the EXIF data. This is achieved through the use of a Custom Tag Handler which must conform to the following protocol:

```
@protocol EXFTagHandler
      -(void)decodeTag:(NSMutableDictionary*) keyedValues: (NSNumber*) tagId:
      (CFDataRef*) tagData: (BOOL) bigEndianOrder;

      -(int) getSizeOfValue:(id)value;
      -(BOOL)supportsValueType:(id) value;
      -(void)encodeTag: (NSMutableData*) targetBuffer: (id) tagData:(BOOL)
      bigEndianOrder;

@optional
      -(int) tagFormat;
      -(int) parentTagId;
      -(BOOL) isEditable;

@end
```

The decode and encode methods are called by the library when it is constructing an instance from the JPEG bytes or adding the byte representation of the data into the JPEG byte format. The sizeof and supportsValueType are used to enable the library to test if an actual value can be supported by the handler and what the encoded byte size of the data would be.

We can see how this would be implemented by examining the internal ASCII tag handler in the library:

```
@implementation EXFASCIIHandler

-(void)decodeTag:(NSMutableDictionary*) keyedValues: (NSNumber*) tagId:
      (CFDataRef*) tagData: (BOOL) bigEndianOrder{
      UInt8* ptr = (UInt8*) CFDataGetBytePtr(*tagData);
      CFIndex byteLength = CFDataGetLength(*tagData);
      NSString* value = [EXFUtils createStringFromBuffer:&ptr: byteLength:
      NSASCIIStringEncoding];
      [keyedValues setObject: value forKey: tagId];
      [value release];
}

-(void)encodeTag: (NSMutableData*) targetBuffer: (id) tagData:(BOOL)
bigEndianOrder{
      // tag data is an array of NSNumber
      int length = [((NSString*)tagData)
      lengthOfBytesUsingEncoding:NSASCIIStringEncoding];
      const char* cString = [((NSString*)tagData)
      cStringUsingEncoding:NSASCIIStringEncoding];
            [targetBuffer appendBytes: cString length:length];
}

-(BOOL)supportsValueType:(id) value{
      if ([value isMemberOfClass:[NSString class]]){
            return TRUE;
      }else{
            return FALSE;
      }
}

-(int) getSizeOfValue:(id)value{
      if ([value isKindOfClass:[NSString class]]){
            return[((NSString*)value)
      lengthOfBytesUsingEncoding:NSASCIIStringEncoding];
      }else{
            return -1;
      }
}

-(NSString*) description{
      return @"EXF ASCII Handler";
}
@end
```

The decodeTag method receives a reference to the keyedTagValues Map, which contains all the decoded tags, the tagid, a CFDataRef (which is really just NSData), containing the bytes specified in the JPEG Image for the tag and an indicator of the endianess nature of the data.

The encode tag is really the reverse of this method and the developer must insert into the target bufefr the byte representation of the tagValue passed in. It is the developers responsibility to comply with the EXIF definition for allowable size and endian values.

The supportsValueType method is called by the library prior to the encode data to allow the handler to reject the data based on its type. Similarly, the getSizeOf must be used by the handler to indicate how large the bytes would be if the tagValue was encoded. If the handler returns a different number of bytes in the encode than it indicates in the sizeof then the library will throw an expeption. These methods are both called prior to the encode method.

The tag handler is registered using the a similar format to the code below and must be set prior to any tag processing if you want to the behaviour to apply to all your processing.

```
[exifMetadata addHandler:asciiHandler :EXIF_ExifVersion];
```

Here we are registering the asciiHandler to handle the tag EXIFVersion.

Retrieving the Modified Image

Once we have made our changes, we must then be able to reconstitute the JPEG with the modified Tags. this is pretty straight forward:

```
-(void) populateImageData: (NSMutableData*) newImageData;
```

The programmer must just pass an instance of NSMutableData to the populateImageData method. this will fill array with the bytes of the changed image which can then be exported to file or used to instantiate a UIImage object.