AOSX

ATIK OSX Drivers

A Developers Guide

Release 0.02 (Monday, 31 December 2012)

Foreword

What's in the tin and in this guide?

These are the first 100% native OSX ATIK drivers presented as two OSX Frameworks - one current and one legacy.

As you're a developer - I'll keep it concise and to the point in this guide, I'll assume you know OSX and Objective-C/C++.

I don't work or have ties with ATIK except the signing of an NDA to share the original Artemis Driver sources. So please direct problems with these drivers to me rather than ATIK - I'll be happy to hear any issues you have with these drivers. However cameras under the Legacy Driver are a lower priority than those in the main driver.

Warrantee & Legal

By using these drivers, you release me from any legal claims or damages relating form the use (either directly or indirectly).

These drivers are provided "as is" and are not provided with any warrantee, guarantee for fitness for purpose or expectation to provide support. You can find me on the excellent Stargazer's Lounge forum as NickK or atikosxdrivers@adrenalin-junkie.net. Note these drivers will only work for ATIK cameras and ATIK devices.

The current Alpha-release of these drivers should be regarded as experimental

and should not be used in production code.

Distribution

These drivers, in their unchanged binary form, are freely distributable.

The source for this driver is covered by NDA and cannot be made available.

What this document covers

This document covers:

Product coverage - which cameras are supported, tested etc.

Architecture - how the driver sits.

Developing - adding to Apple's XCode

Using the API in practice - my first program.

Differences between the existing Artemis DLL.

Known issues and planned allocations of features in releases.

AOSX

Table of Contents

Product Coverage

You'll be pleased to know that the driver supports the majority of the current ATIK products. I have a few cameras that will allow testing, however I don't have the entire catalogue! Although the following are supported, in theory, I have indicated the devices the driver has been verified with.

Current Cameras

Camera	Supported	Tested
3xx series (incl 383L)	Yes	383L Mono
4xx series	Yes	No
4000 series	Yes	Partial 4000 Mono
11000 series	Yes - experimental	No - BETA release.
Titan	Yes	Titan Mono

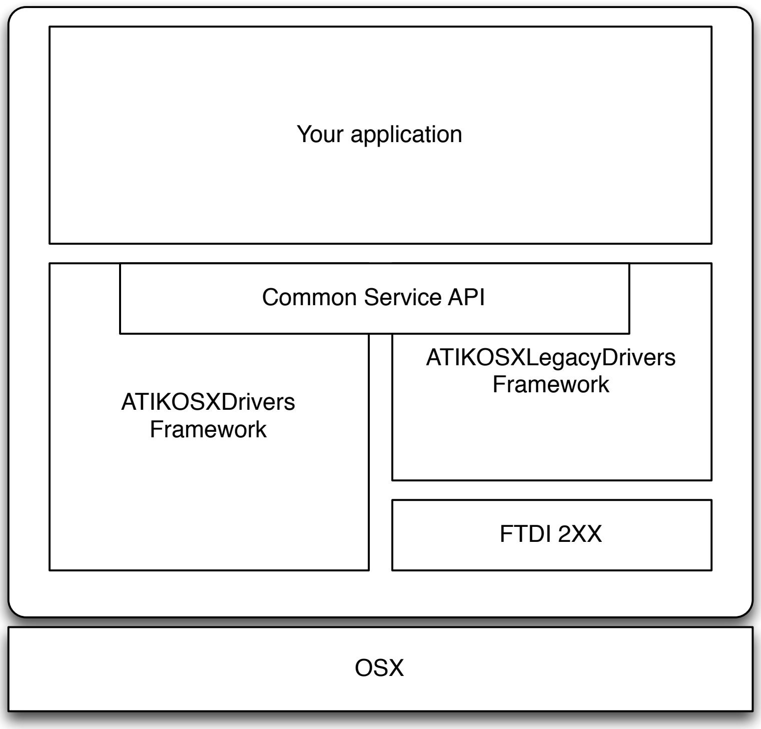
Legacy USB1.1 Cameras

Camera	Supported	Tested
2xx series	Yes	No
16IC series	Yes	16IC Mono

Filter Wheels

Filter Wheel	Supported	Tested
EFW1	Unknown	Unknown
EFW2	Yes	5 slot

Architecture



Common Service API (AOSX)

The common service API allows an application to use one or both of the drivers without knowing the details. Through this API it's possible to discover devices connected. Additionally alerts indicate when connecting & disconnecting should you need it. Applications can claim, release and utilise devices through these interfaces. More later..

ATIKOSXDrivers

This driver supports all the USB 2.0 devices such as cameras and filterwheels that don't use the FTDI library.

ATIKOSXLegacyDrivers

This driver supports all the USB1.1 that would traditionally require the FTDI drivers with the PC Artemis DLL. Infact the FTDI drivers are baked into the driver so you do not have to install them separately. As these are non-VCP drivers they should be fine for use with sandboxing (untested).

In use the only difference between the Legacy and normal drivers is that they're a bit slower.. except for the 16IC support - there's a separate section later for this.

Developing

Binary Finery

Developing in XCode is expected and the drivers are provided as 64bit 10.8 binaries.

Adding to your project

Simply add them to your application by either embed as a private framework within the application bundle -or- use an installer into the Library.

Where did my runloop go?

The libraries do not use any threading, they utilise the thread that calls them to perform any action. If this happens to be your GUI thread then for some operations this could cause your runloop to disappear into the ether for a period of time causing the GUI to lockup during long downloads.

One technique is to use a separate thread to initialise and call the library - until you're more adept. Note that Apple's HID implementation requires the runloop to cycle to perform I/O. Lovely.

Scaling out threading

It is possible to scale out with separate threads/tasks to perform actions on multiple devices simultaneously *and* the preparation and post-processing of images can be scaled for a thread on each image frame.

There is more work to be done on the Apple HID with multiple EFW2s with multiple threads in this area but it's working fine at the moment. The management interfaces should only be called by a single thread.. at the moment they're not thread safe.

This multi-threading allows for multiple cameras to be downloading, deferring processing to later or using additional CPU resources to farm off any post-processing which is useful if you're trying to take lots of images quickly.

Tracing

The Alpha releases will always output tracing automatically to your desktop. If you find a problem - please can you provide this to me.

Using the API - a program

I’m not going to regurgitate the APIs - so here’s a more useful example of it in use.

```
#import "HelloWorld.h"

#import <ATIKOSXDrivers/ATIKOSXDrivers.h>

@implementation HelloWorld

-(void)HelloWorld {

    // Lets initialise..

    // create the drivers - you'll need to keep hold of it..
    ATIKOSXDrivers *drivers=[[ATIKOSXDrivers alloc]init];

    // start the support - at this point the drivers setup USB/HID hooks
    [drivers startSupport];


    // Now any time you want to discover, claim or release devices you'll
    // want the services interface:
    id<Services> services = [drivers serviceManagement];

    // let's see what's attached..
    NSDictionary *available = [services availableServices];

    NSLog(@" The available ATIK devices are: %@", available);

    // Next up let's claim a Titan for our own exclusive use
    id obj = [services claimService:@"ATIK Titan"];
    id<Imager100> imager = obj;

    // let's make a frame ready (prepare) for a 10 second photo.
    // the additional parameters for the init may change so don't use)
    ImageFrame *imageFrame  = [[ImageFrame alloc]initWithSize:0 at:nil];

    imageFrame.duration = 10.0; // set the exposure duration requested.

    // next we prepare - the camera sets up the frame as required
    // (this could be a sperate thread) or you could prepare a set of
    // frames in advance.
    [imager prepare:imageFrame];

    // we're going to tell the frame just to make it's own buffer
    // rather than set the imagebuffer of the size imagebufferSize bytes.
    [imageFrame selfAllocate];

    // let's take the photo - blocking only for now.
    [imager snapShot:imageFrame blocking:YES];

    // do any post processing (this could be done by a seperate thread)
    [imager postProcess:imageFrame];

    // imageFrame.imagebuffer now holds our image.
    // imageFrame.imagebufferSize is still the size of the buffer
    // We could use NSBitmapImageRep to create a TIFF as our titan is mono..
    [self makeMonoTIFF:imageFrame];

    // lets give the titan back..
    [services releaseService:imager];

    // let's shutdown the driver - unhooking
    [drivers endSupport];
}
```

The main things are:

1. Keep hold of the driver instance until you don't need it.

2. The startSupport will setup OSX hooks and immediately start identifying existing connected devices and any device connected from this point onwards.

3. The object returned by claimService could have any interface - this can be checked by conformsToProtocol:

4. The identity used in the beta will be unique - including the serial number or unique identifier.

5. ImageFrame is configured ***before*** calling prepare:

6. After prepare you'll note that the dimensions of the camera etc are present in the frame size etc if no subframe is setup.

7. It's possible to provide your own buffer, or reuse an ImageFrame (use prepare: again before reuse). If you use your own buffer expect data to be modified until after the post processing is complete.

8. Ending support does not destroy the service objects - these must be released. With AOSX these may be held for a period if your application crashed to prevent the device from being reassigned - but that's a different story.

The TIFF routine works for 16bit images.. but you could code up a FITS routine if you wanted..

```
-(void)makeMonoTIFF:(ImageFrame*)imageframe {
    uint8_t *input = (uint8_t*) imageframe.imagebuffer;

    unsigned char *planes[1];
    planes[0]=(unsigned char*)input;

    NSBitmapImageRep *bitmapRep = [[NSBitmapImageRep alloc]
                                     initWithBitmapDataPlanes:planes
                                     pixelsWide:imageframe.width
                                     pixelsHigh:imageframe.height
                                     bitsPerSample:16
                                     samplesPerPixel:1
                                     hasAlpha:NO
                                     isPlanar:NO
                                     colorSpaceName:NSCalibratedWhiteColorSpace
                                     bitmapFormat:0
                                     bytesPerRow:(2 * imageframe.width)
                                     bitsPerPixel:16];
    if( bitmapRep!=nil) {
        NSData *tiff = [bitmapRep representationUsingType:NSTIFFFileType
properties:nil];
        [tiff writeToURL:myURL atomically:YES];
    }
}
```

To allocate a filterwheel, for example, the same steps are taken:

```
- (IBAction)testFWSelectCycleActionTest:(id)sender {
    id<Services> services = [driver serviceManagement];
    NSDictionary *available = [services availableServices];

    NSLog(@"available services: %@", available);

    id serviceObject = [services claimService:@"ATIK EFTW2"];
    if( ![serviceObject conformsToProtocol:@protocol(FilterWheel100)]) {
        // oops
        NSLog(@"Object returned doesn't support FilterWheel100!");
        return;
    }
    id<FilterWheel100> efw =  serviceObject;

    [efw setPosition:0];
    sleep(5);
    [efw setPosition:1];
    sleep(5);
    [efw setPosition:2];
    sleep(5);
    [efw setPosition:3];
    sleep(5);
    [efw setPosition:4];

    [services releaseService:efw];
}
```

To use the Legacy cameras at the same time then all you need todo is initialise an ATIKOSXLegacyDriver class at the same time as the ATIKOSXDriver.

```
    ATIKOSXLegacyDrivers *legacyDriver=[[ATIKOSXLegacyDrivers alloc]init];
    [legacyDriver registerFTDIChipId:0x0ee8f4c9 forUSBSerialNumber:@"A3001QH8"];
    [legacyDriver startSupport];
    id<Services> services = [legacyDriver serviceManagement];
    NSDictionary *available = [legacyDriver availableServices];
```

You'll note the registerFTDIChipId:forUSBSerialNumber: call - this is covered in the later section The 16IC.

After the initialisation just use the legacy device services as you did before (with Imager100 etc).

Differences to Artemis*.DLL

As you've probably noted - the API differs from the Artemis API.

As the innards are covered by NDA there's not much to talk about. However here are the main API differences:

- Objective-C/C++ API in a ***very*** similar to AOSX.
- No threading in the library - 90% of threads in classes cause problems for developers.
- No GUI interactions (as ASCOM seems to like requiring) - there be dragons down that path, especially with Cocoa.
- Image frames are given to the device - the library doesn't create buffers, designed with speed and efficiency.
- Image Frames are prepared, filled and postprocessed.
- No multi-application support from a shared DLL. OSX doesn't have this in user-space drivers.
- All User-mode driver - no Kernel driver components. This prevents blue-screening and is future compatible with OSX.

The similarities with AOSX mean that come the AOSX release it should be simple to slot in the AOSX client framework.

Making a C++ wrapper should be easy, however making a call-for-call DLL API wrapper would be more difficult (especially the multi-app support).

AOSX

The 16IC

Thou battered friend, whose eye has followed the stars for many years - hold my hand.

Supporting the 16IC

FTDI in their infinite wisdom have decided not to bring out a 64bit version of the FTDIChipID library and as Apple have dropped the support for 32bit shared libraries it is now down to the user to use a PC to read the FTDIChipID for their 16IC.

The Legacy Driver allows the application to register a set of mappings for FTDIChipIDs to camera serial number. This allows the driver to use the ID without requiring the use of the defunct FTDI decryption library.

If your application is to support the 16IC then you will need to provide a mechanism to pass this information through.

Obtaining the FTDIChipID

You'll only need to obtain the unique identity number for the user's 16IC once. After that the number can be used for the serial number again and again.

I'll try to get together a PC commandline application to read this using the PC FTDI library. If FTDI bring out a 64bit library then we'll not need to use this manual method.

In the meantime an alternative is to packet sniff on the PC, look for the following:

43 4D 44 0B *C9 F4 E8 0E*

Search for 43 4D 44 0B then take the next 4 bytes (they will differ from the italics - which show my 16IC's IDs).

Registering FTDIChipIDs with serial numbers

And use the following to register:

```
[drivers registerFTDIChipId:0x0ee8f4c9 forUSBSerialNumber:@"A3001QH8"];
```

Some test statistics

I’ve developed and tested the drivers on two platforms:

* MacBook Pro 2.2GHz i7, OSX 10.8

* MacBook Mini 2.66GHz Core 2 Duo, OSX 10.8

The test rig includes two USB ‘channels’:

Channel one - computer connects to a USB 5 meter repeater connected to another 5 meter repeater, connected to a powered USB hub with the following attached: Titan, 4000 carcass, 16IC, EFW2 and a gaming mouse (provides additional bus traffic).

Channel two - computer connects tot a USB 5 meter repeater connected another 5 meter repeater connected via a long standard USB cable to a 383L+.

Tests

There are automated tests that go through all the following permutations and combinations - binning (x&y), sub-frames (top left, borrom left, top right, bottom right quadrants and centred quad), preview mode (on/off),

Performance

The following is with debugging builds:

Titan performance with the above rig has sustained 19.7 frames per second over 50,000 preview bias frames (took 42 minutes). This is faster than ATIK’s 15 frames per second.

The 383L+ downloads in about ~10 seconds - faster than ATIK have quoted at ~12 seconds.

The 16IC support is currently set up less efficiently for debugging purposes in the alpha release.

Release Feature Backlog

Alpha Releases

Features (where camera supports)

- Image exposure
- Image Binning (up to 4 times in X&Y - this is an artificial limit imposed to shorten automated testing time)
- Image sub-framing
- Image preview
- * Setpoint Cooling
- EFW2 Number of Filters
- EFW2 Filter selection

Current Cameras supported (USB 2.0)

- 400-series
- 300-series
- 4000/11000-series - this is experimental, Beta will have this working.
- Titan

Legacy Cameras supported (USB 1.1)

- 200-series
- 16IC

Note 200-series DSO parallel port triggers are not supported in this release.

Release Focus

The focus on this release is to bug fix - I know there are a few already (see known issues). There may be some API updates over subsequent alpha releases.

Beta Releases

Features, where cameras support

* Image precharge

* Image sub-sample

* Image auto-black leveling (Titan)

* Explicit camera disconnect

* Soft Reconnect

* Overlapped exposures (overlapping exposure and download where camera supports)

* Abort exposure

* Abort download

* OSC colour support

* Guide Port control

* GPIO port

Release Focus

This release is focused on adding

Release Notes

Experimental Alpha Release - v0.01 (30 Dec 2012)

* Sub-frames aren’t checked if dimensions aren’t in scope which can cause crashes, this was an oversight. To be fixed in next alpha release.

* Initialisation may stall.

* USB List may hang with EFW2, this is down to the USB get description with 255 characters.

* ImagerFrame allocates more memory than required - this is down to the USB framing of OSX. At the moment it’s a little over-zealous in additional allocation. To be fixed in next alpha release.

* Titan images white out

Experimental Alpha Release - v0.02 (06 Jan 2013)

Fixes

* Tracing file clash for both drivers - instead the alpha drivers will now create separate logs for each driver on the desktop.

* Titan whiteout fixed, titan improvements made across the board..

* Titan and 383L sub-framing fixed

* USB hand on driver’s use of 256 byte descriptor buffer - removed this additional info from usb list currently.

* Initialisation stall along with a few other improvements.

Outstanding

* HIID device disconnect when unplugged

* Memory allocation optimisation

Astronomy on OSX - AOSX

Just to say a little about the fore-mentioned AOSX.

AOSX is an up and coming opensource framework to allow application developers to focus on the application and use the provided services - such as camera driver plugins etc - through a common interface. AOSX is happy if vendors wish to only release closed source plugins or open source the plugin.

The architecture provides a sandboxed environment so if the driver fails, your application wont.

It’s not designed to be a standard - simply an enabler, allowing more apps to support more devices and services to make the Mac a better astronomy platform.

Lastly although the framework is Objective-C/C++ it will support applications and plugins written/ported over in C++.