Image Capture Applications Programming Guide

Carbon > Graphics & Imaging

2007-05-16

Contents

Figures

Introduction

This document describes the Image Capture Architecture, a set of extensions to Mac OS X that enable applications and image capture devices, such as cameras and scanners, to interconnect.

You should read this document if you are writing applications that work with image capture devices such as cameras or scanners, or if you are writing application plug-ins that work with image capture devices, or if you wish to add a new image capture device, such as a camera, scanner, or phone, to the list of supported devices under Mac OS X.

Organization of This Document

This document is organized into the following sections:

- "“Overview” (page 9)”—an overview of the Image Capture Architecture
- "“How to Write an Image Capture Application” (page 13)”— a guide for writing applications that work with image capture devices
- “Working with the SDK and Header Files” (page 27)”— a guide to working with the image capture header files and SDK

See Also

- *Image Capture Applications Reference*
- *Image Capture Device Modules Reference*

Overview

The Image Capture Architecture (part of `Carbon.framework`) is a high-level framework for capturing image data from devices such as scanners, digital cameras, phones and PDAs. The Image Capture interfaces are device independent, so you can use them to gather data from any devices connected to the system. You can get a list of devices, retrieve information about a specific device or its contents, and retrieve files—including images, movies, and sound files—from devices. You can also issue commands to devices that support this feature, to initiate a document scan, take a photo, or delete an image from a camera card, for example.

Mac OS X provides a set of image capture services that enable you to do the following:

- Detect when an image capture device, such as a scanner or camera, has been attached to the host computer via USB or FireWire.
- Launch a browser that detects image capture devices available over Bluetooth or TCP/IP network connections and attach them
- Obtain a detailed list of attached devices and their properties, including directories of images and image metadata
- Download images from devices such as cameras and scan images using scanners
- Detect and use special functions for particular devices, such as the ability to capture a new image or delete existing images
- Write applications that are automatically launched when an appropriate device is attached
- Create device modules for new devices, such as cameras, scanners, phones, and PDAs, so they can be accessed using the Image Capture Architecture framework

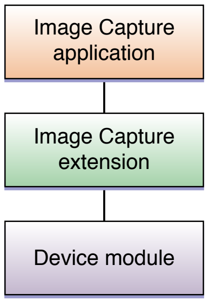
Architecture

There are three pieces to the Image Capture Architecture:

- Image Capture device modules, which are launched when a particular image capture device is attached. If you are adding a new scanner, camera, or other image capture device to Mac OS X, you need to write an image capture device module. The API for writing device modules is part of the Image Capture Device framework (`ICADevice.framework`). (See *Image Capture Device Modules Reference*.)
- Image Capture API functions for cocoa and carbon programmers. These system-level extensions enable applications to work with attached devices at a high level. This API is part of the Image Capture framework (`Carbon/ImageCapture/ImageCapture.h`). It is described in this document and in *Image Capture Applications Reference*.
- `Image Capture Extension.app`, a faceless application that runs in the background and provides the connections between device modules and the applications, shielding applications programmers from having to deal with individual devices, and providing a common interface between device modules and applications.

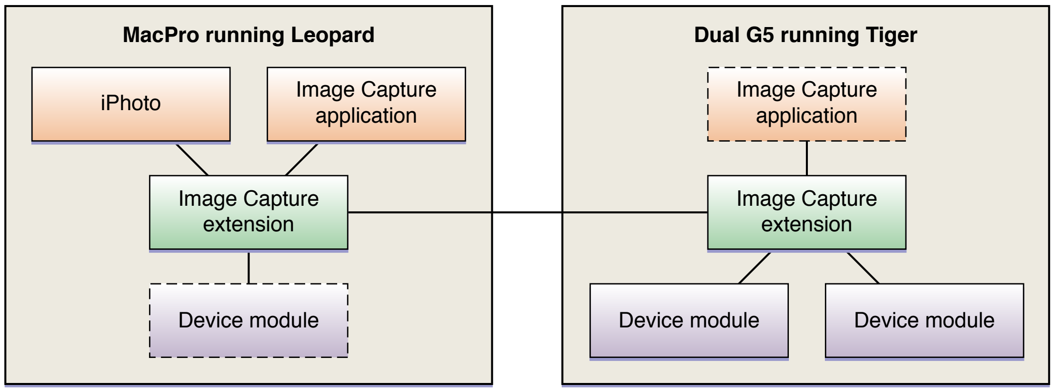
The image capture services architecture is conceptually somewhat similar to TWAIN architecture, with the image capture extension taking the place of the TWAIN data source manager and the the device module taking the place of the TWAIN data source, as shown in Figure 1-1 (page 10).

Figure 1-1 Main Components of the Image Capture Architecture



The Image Capture applications, extension, and device modules are all run in separate processes, making it possible to dynamically add device modules into the mix, share devices among applications, access devices from applications running on separate host computers, and even access device modules from applications running on different CPU architectures, as shown in Figure 1-2 (page 10).

Figure 1-2 Image Capture Architecture



As illustrated, it is possible for applications on one computer to access device modules running on another computer. It is also possible for multiple applications to access multiple devices, and for multiple applications to share access to a single device.

Sequence of Events

The process operates as follows: when a device is connected to the host computer, via FireWire or USB, or an image-capture device is attached from a Bluetooth or TCP/IP network connection, the operating system searches among the installed device modules for the most suitable one, using the device module's associated `DeviceInfo.plist`. (If more than one module is installed for a given device, the system selects the highest `CFBundle` version number, taken from the `Info.plist` file.)

If an application is configured to be launched when this type of device is detected, that application is launched. For example, connecting a digital camera or pushing the "scan" button on a scanner can invoke your application without user intervention.

Note: In order to have your application launched when a device is attached, instruct your end user to run the Image Capture application and select your application in the dialog box presented by the Image Capture > Preferences menu.

When launched, your application obtains a device list object from the Image Capture API, chooses a device from the list, and obtains the dictionary for that device. The dictionary contains a list of objects associated with the device, such as images, movies, audio files, and directories.

The dictionary for the device also includes information about the device, such as whether it supports capturing images on command or deleting images. You can use the Image Capture API to issue such commands to the device.

Your application can download the objects of interest directly or obtain a dictionary of metadata specific to individual objects to determine whether they should be downloaded.

Note: Earlier versions of image capture services required the application to iterate through the device list to determine each device type, then iterate through the objects contained on a device, such as images, audio files, and directories. The API for this earlier method is still in the headers for backward compatibility, but the new API, using dictionaries rather than iterating through the object hierarchy, is recommended, and is the API documented in the following chapter.

Image capture applications should register for notification of image capture events, such as the addition of new devices, the removal of existing devices, or the change of storage in an existing device, such as the insertion of a new memory card into a mounted camera or card reader.

TWAIN Compatibility

Image capture devices can be accessed using the native image capture API described in this document, or they can be accessed as TWAIN data sources. In order for devices to be recognized by TWAIN applications as well as Image Capture applications, the device manufacturer must provide both a TWAIN data source module and an Image Capture device module. If both are available, the system uses the Image Capture device module by default, and the TWAIN data source when needed.

Device Browser

An Image Capture application has access to all attached image capture devices. USB and FireWire devices are automatically attached when they are connected, but Bluetooth and TCP/IP network devices may not be. The Image Capture API includes a device browser that your application can invoke, allowing the end user to browse for available devices and attach them. Once mounted, they appear in the device list object when it is next obtained. Device manufacturers should use Bonjour protocol to advertise available TCP/IP network devices.

How to Write an Image Capture Application

You can write a complete image capture application using a small subset of the image capture API.

For most applications, these six function calls will be all you really need:

```
ICAGetDeviceList
ICARegisterForEventNotification
ICACopyObjectPropertyDictionary
ICAOpenSession
ICAObjectDownLoadFile
ICACloseSession
```

An Image Capture Application in Seven Steps

You can write an image capture application in seven steps.

1. Get the device object list by calling `ICAGetDeviceList`.
2. Call `ICACopyObjectPropertyDictionary` to get the properties of the attached devices.
3. Register for notifications on the devices you are interested in using `ICARegisteForEventNotification`.
4. Open a session on a specific device by calling `ICAOpenSession`.
5. Call `ICACopyObjectPropertyDictionary` again for the device you are interested in, then for each object of interest.
6. Call `ICAObjectDownLoadFile` to get each object that you want.
7. When you are done, call `ICACloseSession`.

This is all you generally need to download images from an attached camera on a USB or FireWire port.

To support more ambitious applications, the Image Capture API includes a device browser that you can invoke to scan for other available devices, such as TCP/IP networked cameras or scanners implementing Bonjour, or devices connected to other networked computers that you can attach indirectly. These devices do not appear in the ICA device list until they are attached through the device browser.

The Image Capture Architecture also includes an application, `Image Capture.app`. In addition to serving as an example application that can download and display images and work with scanners, it allows users to choose what application to launch when a camera is attached or a button is pressed on a scanner. Your end-user documentation should direct users to the Image Capture application in order to make your own application the default when a device is attached or a button is pressed.

Be aware that all of the functions in the Image Capture Architecture API can be called either synchronously or asynchronously. That means that you can either wait while the function executes or pass-in a completion function that will be called when the function completes. Setting up a completion function requires a bit more work, but it is highly recommended, as some of the functions can take a long time to complete. Executing these functions asynchronously does not block the user interface.

The rest of this section goes over the steps in more detail. Each step includes a code snippet that shows how to set up the call asynchronously.

Getting Started

The first step in writing an image capture application is to download the Image Capture SDK (http://developer.apple.com/sdk/). This SDK includes sample code, debugging tools, and example applications and modules for both Cocoa and Carbon developers. The image capture sdk includes a Tools folder, in which you can find an application named `ICAAPITest`. Using this application, you can execute and of the commands documented in this guide. You can also use the application to generate cut-and-paste code in C or objective C, synchronously or asynchronously, for use in your application.

Note: Be sure to attach an image capture device, such as a camera, to your host computer before using the `ICAAPITest` application. Many of the functions will return null or an error if there are no attached devices.

Getting the Device List Object

The first step in an Image Capture application is obtaining a device list object—an object that contains a list of all the connected image capture devices. The following code snippet shows how to call `ICAGetDeviceList` asynchronously.

```
// ------------------------------------------------------- getDeviceListCallback
void getDeviceListCallback (ICAHeader* pbHeader)

{
    ICAGetDeviceListPB * pbPtr = (ICAGetDeviceListPB *)pbHeader;
    if (noErr != pbPtr->header.err)
    {
        // handle error
    } else
    {
        // pbPtr->object   // ICAObject
    }
}
// --------------------------------------------------------------- getDeviceList
void getDeviceList ()
{
    OSErr err;
    ICAGetDeviceListPB pb = {};

    err = ICAGetDeviceList(&pb, getDeviceListCallback);
    if (noErr != err)
    {
        // handle error
    }
}
```

`ICAGetDeviceList` returns a device list object, which is simply an ID code.

Getting the Device List Dictionary

Once you have a device list object, you obtain the properties of the devices using the `ICACopyObjectPropertyDictionary` function.

First, call `ICACopyObjectPropertyDictionary`, passing in the device list object ID. The code snippet below shows how to make the call asynchronously.

```
// ---------------------------------------- copyObjectPropertyDictionaryCallback
void copyObjectPropertyDictionaryCallback (ICAHeader* pbHeader)
{
   ICACopyObjectPropertyDictionaryPB * pbPtr = (ICACopyObjectPropertyDictionaryPB
 *)pbHeader;
    if (noErr != pbPtr->header.err)
    {
        // handle error
    } else
    {
        // pbPtr->theDict   // CFDictionaryRef *
    }
}

// ------------------------------------------------ copyObjectPropertyDictionary
void copyObjectPropertyDictionary ()
{
    OSErr err;
    ICACopyObjectPropertyDictionaryPB pb = {};
    pb.object = <#ICAObject object#>; //device list ID
    err = ICACopyObjectPropertyDictionary(&pb,
copyObjectPropertyDictionaryCallback);
    if (noErr != err)
    {
        // handle error
    }
}
```

This returns a dictionary that includes all the attached devices and some of their properties, such as device name and type. For example, if a camera or memory card reader is attached to the USB port, it will show up as a device in the dictionary. The code snippet below shows an example of the dictionary returned when the device list consists of a single camera attached to the USB port. The object ID of the device is the `icao` property.

```
{
    devices = (
        {
            IORegPath =
"IOService:/MacRISC4PE/ht@0,f2000000/AppleMacRiscHT/pci@8/IOPCI2PCIBridge/usb@B,1/AppleUSBOHCI/CameraCo
 DigiSnap ZZ0 Digital Camera@2b100000/IOUSBInterface@0";
            arch = ppc;
            clients = ();
            "device module" = "/System/Library/Image
Capture/Devices/PTPCamera.app";
            "device type" = camera;
            file = 1668117089;
            icao = 1815C00;
            idProduct = 1427;
            idVendor = 1034;
            ifil = "ZZO Digital Camera"; ]
            locationID = 722468864;
            remote = NO;
            thuP = 3573472;
}
    );
}
```

Notice that the dictionary contains the device type ("camera") and the USB vendor and product ID. It also contains an object ID for each device, shown here in the `icao` field.

Notifications

If you want your application to be notified when a device is added or removed, you should register for notifications. This allows you to detect a camera being unplugged, for example, or a new memory card being inserted into a card reader. The function call to register for notifications is `ICARegisterForEventNotification`. You can register for all devices, of for a specific device, using the `icao` of the device from the device list dictionary. The following code snippet shows how to make the call asynchronously.

```
// ------------------------------------------- registerEventNotificationCallback
/* This function is called to notify the client of events for which the client
 has registered interest */
static void notificationCallback( CFStringRef notificationType, CFDictionaryRef
 notificationDictionary )
{
    /* handle notifications here */
}
/* This function is used if ICARegisterForEventNotification() called
asynchronously. */
void registerForEventNotificationCallback( ICAHeader* pbHeader )
{
    ICARegisterForEventNotificationPB * pbPtr = (ICARegisterEventNotificationPB
 *)pbHeader;
    if (noErr != pbPtr->header.err)
    {
        // handle error
    } else
    {
        // no return value(s)
    }
}
/* Sample code snippet to illustrate the use of ICARegisterForEventNotification()
 */
OSErr registerForNotifications()
{
    OSErr                              err = noErr;
    ICARegisterForEventNotificationPB  pb = {};
    CFStringRef                        notificationsOfInterest[]  = {
```

```
kICANotificationTypeDeviceRemoved,

kICANotificationTypeDeviceInfoChanged,

kICANotificationTypeDeviceWasReset,

kICANotificationTypeCaptureComplete,

kICANotificationTypeTransactionCanceled,

kICANotificationTypeStoreAdded,

kICANotificationTypeStoreRemoved,

kICANotificationTypeStoreFull,

kICANotificationTypeStoreInfoChanged,

kICANotificationTypeObjectAdded,

kICANotificationTypeObjectRemoved,

kICANotificationTypeObjectInfoChanged
                                                                 };
    CFMutableArrayRef  array = CFArrayCreate( kCFAllocatorDefault, (const
void**)&notificationsOfInterest, 12, &kCFTypeArrayCallBacks );
    pb.header.refcon    = (long)self;
    pb.objectOfInterest = <#ICAObject object#>
/* valid values are 0, device list object, or device object. 0 gets notifications
 related to any object. */
    pb.eventsOfInterest = (CFArrayRef)array;
    pb.notificationProc = notificationCallback;
    pb.options          = NULL;
    err = ICARegisterForEventNotification( &pb,
registerForEventNotificationCallback );
    CFRelease( array );
    return err;
}
```

Note: Be sure to use the `ICARegisterForEventNotification` function, not the older, deprecated `ICARegisterEventNotification` function. The names are confusingly similar.

Opening an ICA Session

It is strongly recommended that you open an ICA session before accessing a device. The following code listing shows how to open an ICA session asynchronously. Pass in the ID of the device you are going to access, as returned in the device list dictionary.

```
// ---------------------------------------------------------- openSessionCallback
void openSessionCallback (ICAHeader* pbHeader)
{
    ICAOpenSessionPB * pbPtr = (ICAOpenSessionPB *)pbHeader;
   if (noErr != pbPtr->header.err)
   {
        // handle error
    } else
    {
        // pbPtr->sessionID   // UInt32
    }
}

// ---------------------------------------------------------------- openSession
void openSession ()
{
    OSErr err;
    ICAOpenSessionPB pb = {};

    pb.deviceObject = <#ICAObject deviceObject#>;
    err = ICAOpenSession(&pb, openSessionCallback);
    if (noErr != err)
    {
        // handle error
    }
}
```

Getting the Object Dictionaries

Choose a device you are interested in from the returned dictionary, and call `ICACopyObjectPropertyDictionary` a second time, this time passing in the object ID of the device you are interested in. The code is identical to the call to obtain the device list dictionary, but this time you would pass the `icao` of the camera, phone, or scanner, rather than the object ID of the device list.

```
// ---------------------------------------- copyObjectPropertyDictionaryCallback
void copyObjectPropertyDictionaryCallback (ICAHeader* pbHeader)
{
    ICACopyObjectPropertyDictionaryPB * pbPtr = (ICACopyObjectPropertyDictionaryPB
 *)pbHeader;
    if (noErr != pbPtr->header.err)
    {
        // handle error
    } else
    {
        // pbPtr->theDict   // CFDictionaryRef *
    }
}

// ------------------------------------------------ copyObjectPropertyDictionary
void copyObjectPropertyDictionary ()
{
    OSErr err;
    ICACopyObjectPropertyDictionaryPB pb = {};
    pb.object = 1815C00; //device object ID
    err = ICACopyObjectPropertyDictionary(&pb,
copyObjectPropertyDictionaryCallback);
    if (noErr != err)
    {
        // handle error
    }
}
```

This returns a dictionary that includes all the interesting objects (such as images, movies, audio files, and directories) associated with the device, along with some information about the objects. For example, if you choose a camera, all the images in the camera's memory card are listed, along with their type (such as JPEG, TIFF, or RAW) and date of capture. The code snippet below shows an example dictionary of an attached camera with two JPEG images.

The dictionary has two main sections. The first section, denoted "`data =`," is a flattened view of the contents of the object, including only files that you are likely to want: images, movies, and audio files. The second section, denoted "`tree =`," is a complete listing of the object's contents, including directory structures and all files on the device, including non-media files. For most applications, the first section provides all the information you need without requiring you to parse directories or extraneous files.

Note: Some of the properties have immediately identifiable names, such as `isiz` or `lock`, while other properties have numeric names, such as the EXIF tag number associated with an image property. The meaning of the properties with numeric designators can be found in the header file.

```
{      IORegPath =
"IOService:/MacRISC4PE/ht@0,f2000000/AppleMacRiscHT/pci@8/IOPCI2PCIBridge/usb@B,1/AppleUSBOHCI/CameraCo
 Digicam ZZ0 Digital Camera@2b100000/IOUSBInterface@0";
    arch = ppc;
    capa = (1684368433, 1935895659, 1919118443);
    clients = ("/Volumes/ImageCapture_Tiger_SDK/Tools/ICAAPITest.app");
    data = (
        {
            0100 = 3264;
            0101 = 2448;
            9003 = "2007:05:04 23:17:04";
            9004 = "2007:05:04 23:17:04";
            datP = 3576096;
            file = 1768776039;
            icao = 3211104;
            ifil = "100_4150.JPG";
            isiz = 2402504;
            lock = 0;
            raw = 0;
            thuP = 3381488;
            tsiz = 49166;
        },
        {
            0100 = 3264;
            0101 = 2448;
            9003 = "2007:05:04 23:17:24";
            9004 = "2007:05:04 23:17:24";
            datP = 3205008;
            file = 1768776039;
            icao = 3205184;
            ifil = "100_4151.JPG";
            isiz = 2263052;
            lock = 0;
            raw = 0;
            thuP = 3379680;
            tsiz = 49166;
        }
    );
    "device module" = "/System/Library/Image Capture/Devices/PTPCamera.app";
```

```
   "device properties" = {"battery level" = 100; "date time" = 20070508T050800;
 };
    file = 1668117089;
    icao = 25254912;
    idProduct = 1427;
    idVendor = 1034;
    ifil = "ZZO Digital Camera";
    locationID = 722468864;
    lock = 0;
    thuP = 3194592;
    tree = (
        {
            datP = 0;
            file = 1684632165;
            icao = 3190416;
            ifil = 0x00010001;
            isiz = 0;
            lock = 0;
            raw = 0;
            thuP = 0;
            tree = (
                {
                    datP = 0;
                    file = 1684632165;
                    icao = 3237280;
                    ifil = DCIM;
                    isiz = 0;
                    lock = 0;
                    raw = 0;
                    thuP = 0;
                    tree = (
                        {
                            datP = 0;
                            file = 1684632165;
                            icao = 3190816;
                            ifil = 100KP880;
                            isiz = 0;
                            lock = 0;
                            raw = 0;
                            thuP = 0;
                            tree = (
                                {
                                    0100 = 3264;
                                    0101 = 2448;
                                    9003 = "2007:05:04 23:17:04";
                                    9004 = "2007:05:04 23:17:04";
                                    datP = 3576096;
                                    file = 1768776039;
                                    icao = 3211104;
                                    ifil = "100_4150.JPG";
                                    isiz = 2402504;
                                    lock = 0;
                                    raw = 0;
                                    thuP = 3381488;
                                    tsiz = 49166;
                                },
                                {
                                    0100 = 3264;
                                    0101 = 2448;
                                    9003 = "2007:05:04 23:17:24";
                                    9004 = "2007:05:04 23:17:24";
                                    datP = 3205008;
                                    file = 1768776039;
                                    icao = 3205184;
                                    ifil = "100_4151.JPG";
                                    isiz = 2263052;
                                    lock = 0;
                                    raw = 0;
                                    thuP = 3379680;
                                    tsiz = 49166;
                                }
                            );
                            tsiz = 0;
                        }
                    );
                    tsiz = 0;
                }
            );
            tsiz = 0;
        },
        {
            datP = 0;
            file = 1684632165;
            icao = 3241360;
            ifil = 0x00020001;
            isiz = 0;
            lock = 0;
            raw = 0;
            thuP = 0;
            tree = (
                {
                    datP = 3197888;
                    file = 1869899877;
                    icao = 3199792;
                    ifil = "DCEMAIL.ABK";
                    isiz = 118;
                    lock = 0;
                    raw = 0;
                    thuP = 3313728;
                    tsiz = 0;
                }
            );
            tsiz = 0;
        }
    );
    tsiz = 9229;
}
```

At this point you can either download the objects of interest immediately or choose the object you are interested in and, once again, call `ICACopyObjectPropertyDictionary`, this time passing in the object ID of the image (or movie, or audio file), which is returned in the `icao` field. The code for obtaining the dictionary is the same, but you pass in `icao` of the object instead of the device.

This returns a dictionary that includes all the metadata associated with the object, such as camera settings, height, width, data size, and a thumbnail. The snippet below shows the metadata that might be returned for a JPEG image. Again, some properties have immediately recognizable names, while others have numeric designators (typically EXIF tag numbers for image properties), whose meaning can be looked-up in the header file.

```
{
    0100 = 3264;
    0101 = 2448;
    0102 = 8;
    0112 = 1;
    011A = 230;
    829A = 0.06666667014360428;
    829D = 4;
    8822 = 2;
    8827 = (100);
    9000 = 0221;
    9003 = "2007:05:04 23:17:05";
    9004 = "2007:05:04 23:17:05";
    9201 = 4;
    9202 = 4;
    9204 = 0;
    9205 = 4;
    9207 = 5;
    9209 = 24;
    920A = 16.79999923706055;
    A000 = 0100;
    A001 = 1;
    A002 = 3264;
    A003 = 2448;
    A217 = 2;
    A401 = 0;
    A402 = 0;
    A403 = 0;
    A406 = 0;
    datP = 3199536;
    file = 1768776039;
    icao = 3350416;
    ifil = "100_4150.JPG";
    isiz = 2402504;
    lock = 0;
    raw = 0;
    rawImage = 0;
    thuP = 3205008;
    tsiz = 49166;
    "\U00a9mak" = "CameraCo";
    "\U00a9mod" = "CameraCo ZZ0 DIGITAL CAMERA";
}
```

Getting the Object

The object dictionary provides the information you need to allocate a data buffer to download the object by calling `ICAObjectDownloadFile`, passing in the object ID from the `icao` property. You can pass in flags to do things such as delete the file after download, change the date in the finder to the capture date of the file, and so on. The code snippet below illustrates a simple download which does none of these things. Use the `ICAAPITest` application to generate code snippets that set the available flags.

```
// ------------------------------------------------------- downloadFileCallback
void downloadFileCallback (ICAHeader* pbHeader)
{
    ICADownloadFilePB * pbPtr = (ICADownloadFilePB *)pbHeader;
    if (noErr != pbPtr->header.err)
    {
        // handle error
    } else
    {
        // pbPtr->fileFSRef   // FSRef *
    }
}
// --------------------------------------------------------------- downloadFile
void downloadFile ()
{
    OSErr err;
    ICADownloadFilePB pb = {};
    pb.fileCreator   = <#OSType fileCreator#>;
    pb.fileType      = <#OSType fileType#>;
    pb.rotationAngle = <#Fixed rotationAngle#>;
    pb.object        = <#ICAObject object#>;
    pb.flags         = <#UInt32 flags#>;
    pb.dirFSRef      = <#FSRef * dirFSRef#>;
    err = ICADownloadFile(&pb, downloadFileCallback);
    if (noErr != err)
    {
        // handle error
    }
}
```

You may also wish to download the object thumbnail to display while the object itself downloads. To do this, call `ICACopyObjectThumbnail`, passing in the object ID of the image, as shown in the following code snippet.

```
// ------------------------------------------------ copyObjectThumbnailCallback
void copyObjectThumbnailCallback (ICAHeader* pbHeader)
{
    ICACopyObjectThumbnailPB * pbPtr = (ICACopyObjectThumbnailPB *)pbHeader;
    if (noErr != pbPtr->header.err)
    {
        // handle error
    } else
    {
        // pbPtr->thumbnailData   // CFDataRef *
    }
}
// -------------------------------------------------------- copyObjectThumbnail
void copyObjectThumbnail ()
```

```
{
    OSErr err;
    ICACopyObjectThumbnailPB pb = {};
    pb.thumbnailFormat = <#OSType thumbnailFormat#>;
    pb.object          = <#ICAObject object#>;
    err = ICACopyObjectThumbnail(&pb, copyObjectThumbnailCallback);
    if (noErr != err)
    {
        // handle error
    }
}
```

Closing the Session

When you are done accessing a device, and before your application terminates, you should close the ICA session. Closing the ICA session lets the image capture extension know that your application is no longer available to execute pending completion functions, or to receive notifications. Pass in the device ID as you did when opening the session.

```
// ------------------------------------------------------- closeSessionCallback
void closeSessionCallback (ICAHeader* pbHeader)
{
    ICACloseSessionPB * pbPtr = (ICACloseSessionPB *)pbHeader;
    if (noErr != pbPtr->header.err)
    {
        // handle error
    } else
    {
        // no return value(s)
    }
}
// --------------------------------------------------------------- closeSession
void closeSession ()
{
    OSErr err;
    ICACloseSessionPB pb = {};
    pb.sessionID = <#UInt32 sessionID#>;
    err = ICACloseSession(&pb, closeSessionCallback);
    if (noErr != err)
    {
        // handle error
    }
}
```

Opening a Device Browser

The device list object contains a list of attached devices. USB and FireWire devices attach when connected, but Bluetooth and TCP/IP network devices need to be attached by the user before they appear in the device list. To invoke a device browser that allows your end-user to browse for available devices and attach them, call `ICAShowDeviceBrowser`, as shown in the code snippet below.

```
// ------------------------------------------ show device browser
void showDeviceBrowser ()
```

```
{
OSErr err;
err = ICAShowDeviceBrowser( null );
}
```

Working with the SDK

The software development kit (sdk) contains several helpful items, including tools and sample applications. Developers should also be aware that the header files include a number of deprecated functions that are present for backwards compatibility, and should not be used in new applications.

Files and Folders

The SDK includes folders for device modules, documentation, sample applications, tools, and TWAIN modules and applications.

Tools Folder

All developers should check out the tools folder. It contains tools that will help you to write code, debug your applications and device modules, and understand the API. In particular, you should run the `ICAAPITest` application to generate code snippets and test the results of various functions when different devices are connected.

Applications Folder

Applications developers should look in the sample applications folder for applications similar to the ones you wish to develop.

If you are writing a device module, the `SimplePTPCamera` sample application should serve as a template to guide you.

TWAIN Folder

The TWAIN folder is useful for developing TWAIN modules and plug-ins that are compatible with Mac OS X.

Header Files

The `ICAApplication.h` header file contains a number of deprecated functions that are there for backward compatibility. These functions are marked as deprecated in the *Image Capture Applications Reference*. You don't need them, and you shouldn't use them in new applications. Don't be confused by their presence. For image capture applications, just stick to the API described in this document.

There are functions in the `ICAApplication.h`header file that are not described in this document that you will need in order to work with scanners. There is also a separate header file for writing device modules, `ICADevices.h`. You are advised to obtain a paid ADC membership and consult with Apple engineering directly if you need assistance with writing scanner software or writing a device module.

Document Revision History

This table describes the changes to *Image Capture Applications Programming Guide*.

Date	Notes
2007-05-16	New document describing how applications interact with image capture devices such as cameras and scanners.