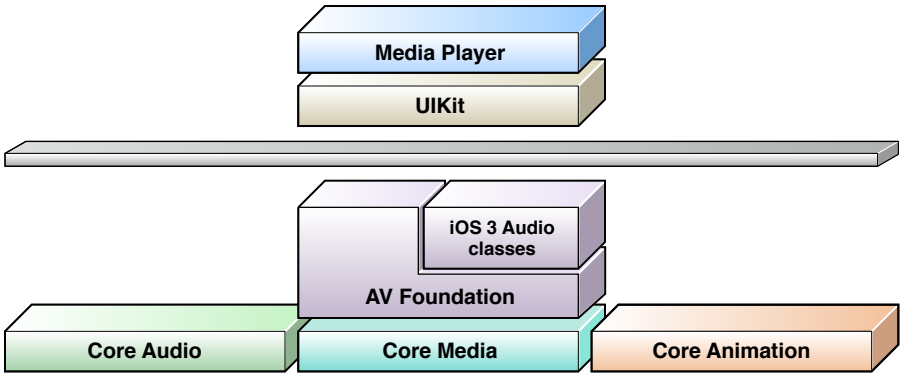
AV Foundation Programming Guide

Developer

Contents

About the AV Foundation Framework

AV Foundation is one of several frameworks that you can use to play and create time-based audiovisual media. It provides an Objective-C interface you use to work on a detailed level with time-based audiovisual data. For example, you can use it to examine, create, edit, or reencode media files. You can also get input streams from devices and manipulate video during realtime capture and playback.



You should typically use the highest-level abstraction available that allows you to perform the tasks you want. For example, in iOS:

- If you simply want to play movies, you can use the Media Player Framework (`MPMoviePlayerController` or `MPMoviePlayerViewController`), or for web-based media you could use a `UIWebView` object.
- To record video when you need only minimal control over format, use the UIKit framework (`UIImagePickerController`).

Note, however, that some of the primitive data structures that you use in AV Foundation—including time-related data structures and opaque objects to carry and describe media data—are declared in the Core Media framework.

AV Foundation is available in iOS 4 and later, and OS X 10.7 and later. This document describes AV Foundation as introduced in iOS 4.0. To learn about changes and additions to the framework in subsequent versions, you should also read the appropriate release notes:

- *AV Foundation Release Notes* describe changes made for iOS 5.
- *AV Foundation Release Notes (iOS 4.3)* describe changes made for iOS 4.3 and included in OS X 10.7.

AV Foundation is an advanced Cocoa framework. To use it effectively, you must have:

- A solid understanding of fundamental Cocoa development tools and techniques
- A basic grasp of blocks
- A basic understanding of key-value coding and key-value observing
- For playback, a basic understanding of Core Animation (see *Core Animation Programming Guide*)

Relevant Chapters "Time and Media Representations" (page 55)

Representing and Using Media with AV Foundation

The primary class that the AV Foundation framework uses to represent media is `AVAsset`. The design of the framework is largely guided by this representation. Understanding its structure will help you to understand how the framework works. An `AVAsset` instance is an aggregated representation of a collection of one or more pieces of media data (audio and video tracks). It provides information about the collection as a whole, such as its title, duration, natural presentation size, and so on. `AVAsset` is not tied to particular data format. `AVAsset` is the superclass of other classes used to create asset instances from media at a URL (see "Using Assets" (page 9)) and to create new compositions (see "Editing" (page 7)).

Each of the individual pieces of media data in the asset is of a uniform type and called a **track**. In a typical simple case, one track represents the audio component, and another represents the video component; in a complex composition, however, there may be multiple overlapping tracks of audio and video. Assets may also have metadata.

A vital concept in AV Foundation is that initializing an asset or a track does not necessarily mean that it is ready for use. It may require some time to calculate even the duration of an item (an MP3 file, for example, may not contain summary information). Rather than blocking the current thread while a value is being calculated, you ask for values and get an answer back asynchronously through a callback that you define using a block.

Relevant Chapters "Using Assets" (page 9)

"Time and Media Representations" (page 55)

Playback

AVFoundation allows you to manage the playback of asset in sophisticated ways. To support this, it separates the presentation state of an asset from the asset itself. This allows you to, for example, play two different segments of the same asset at the same time rendered at different resolutions. The presentation state for an asset is managed by a **player item** object; the presentation state for each tracks within an asset is managed

by a **player item track** objects. Using the player item and player item tracks you can, for example, set the size at which the visual portion of the item is presented by the player, set the audio mix parameters and video composition settings to be applied during playback, or disable components of the asset during playback.

You play player items using a **player** object, and direct the output of a player to Core Animation layer. On iOS 4.1 and later, you can use a **player queue** to schedule playback of a collection of player items in sequence.

Relevant Chapters "Playback" (page 19)

Reading, Writing, and Reencoding Assets

AV Foundation allows you to create new representations of an asset in several ways. You can simply reencode an existing asset, or—on iOS 4.1 and later—you can perform operations on the contents of an asset and save the result as a new asset.

You use an **export session** to reencode an existing asset into a format defined by one of a small number of commonly-used presets. If you need more control over the transformation, on iOS 4.1 and later you can use an **asset reader** and **asset writer** object in tandem to convert an asset from one representation to another. Using these objects you can, for example, choose which of the tracks you want to be represented in the output file, specify your own output format, or modify the asset during the conversion process.

To produce a visual representation of the waveform, you use an asset reader to read the audio track of an asset.

Relevant Chapters "Using Assets" (page 9)

Thumbnails

To create thumbnail images of video presentations, you initialize an instance of `AVAssetImageGenerator` using the asset from which you want to generate thumbnails. `AVAssetImageGenerator` uses the default enabled video track(s) to generate images.

Relevant Chapters "Using Assets" (page 9)

Editing

AV Foundation uses **compositions** to create new assets from existing pieces of media (typically, one or more video and audio tracks). You use a mutable composition to add and remove tracks, and adjust their temporal orderings. You can also set the relative volumes and ramping of audio tracks; and set the opacity, and opacity ramps, of video tracks. A composition is an assemblage of pieces of media held in memory. When you export a composition using an **export session**, it's collapsed to a file.

On iOS 4.1 and later, you can also create an asset from media such as sample buffers or still images using an **asset writer**.

Media Capture and Access to Camera

Recording input from cameras and microphones is managed by a **capture session**. A capture session coordinates the flow of data from input devices to outputs such as a movie file. You can configure multiple inputs and outputs for a single session, even when the session is running. You send messages to the session to start and stop data flow.

In addition, you can use an instance of **preview layer** to show the user what a camera is recording.

Relevant Chapters "Media Capture" (page 33)

AV Foundation's Audio-Related Classes

There are two facets to the AV Foundation framework—API related just to audio, which was available prior to iOS 4; and API introduced in iOS 4 and later. The older audio-related classes provide easy ways to deal with audio. They are described in *Multimedia Programming Guide*, not in this document.

- To play sound files, you can use `AVAudioPlayer`.
- To record audio, you can use `AVAudioRecorder`.

You can also configure the audio behavior of your application using `AVAudioSession`; this is described in *Audio Session Programming Guide*.

Concurrent Programming with AV Foundation

Callouts from AV Foundation—invocations of blocks, key-value observers, or notification handlers—are not guaranteed to be made on any particular thread or queue. Instead, AV Foundation invokes these handlers on threads or queues on which it performs its internal tasks. You are responsible for testing whether the thread or queue on which a handler is invoked is appropriate for the tasks you want to perform. If it's not (for example, if you want to update the user interface and the callout is not on the main thread), you must redirect the execution of your tasks to a safe thread or queue that you recognize, or that you create for the purpose.

If you're writing a multithreaded application, you can use the `NSThread` method `isMainThread` or `[[NSThread currentThread] isEqual:<#A stored thread reference#>]` to test whether the invocation thread is a thread you expect to perform your work on. You can redirect messages to appropriate threads using methods such as `performSelectorOnMainThread:withObject:waitUntilDone:` and `performSelector:onThread:withObject:waitUntilDone:modes:`. You could also use `dispatch_async(3) Mac OS X Manual Page` to "bounce" to your blocks on an appropriate queue, either the main queue for UI tasks or a queue you have up for concurrent operations. For more about concurrent operations, see *Concurrency Programming Guide*; for more about blocks, see *Blocks Programming Topics*.

Using Assets

Asset can come from a file or from media in the user's iPod Library or Photo library. Simply creating an asset object, though, does not necessarily mean that all the information that you might want to retrieve for that item is immediately available. Once you have a movie asset, you can extract still images from it, transcode it to another format, or trim the contents.

Creating an Asset Object

To create an asset to represent any resource that you can identify using a URL, you use `AVURLAsset`. The simplest case is creating an asset from a file:

```
NSURL *url = <#A URL that identifies an audiovisual asset such as a movie file#>;
AVURLAsset *anAsset = [[AVURLAsset alloc] initWithURL:url options:nil];
```

Options for Initializing an Asset

`AVURLAsset`'s initialization methods take as their second argument an options dictionary. The only key used in the dictionary is `AVURLAssetPreferPreciseDurationAndTimingKey`. The corresponding value is a boolean (contained in an `NSValue` object) that indicates whether the asset should be prepared to indicate a precise duration and provide precise random access by time.

Getting the exact duration of an asset may require significant processing overhead. Using an approximate duration is typically a cheaper operation and sufficient for playback. Thus:

- If you only intend to play the asset, either pass `nil` instead of a dictionary, or pass a dictionary that contains the `AVURLAssetPreferPreciseDurationAndTimingKey` key and a corresponding value of `NO` (contained in an `NSValue` object).
- If you want to add the asset to a composition (`AVMutableComposition`), you typically need precise random access. Pass a dictionary that contains the `AVURLAssetPreferPreciseDurationAndTimingKey` key and a corresponding value of `YES` (contained in an `NSValue` object—recall that `NSNumber` inherits from `NSValue`):

```
NSURL *url = <#A URL that identifies an audiovisual asset such as a movie
 file#>;
NSDictionary *options = [NSDictionary dictionaryWithObject:[NSNumber
numberWithBool:YES]

forKey:AVURLAssetPreferPreciseDurationAndTimingKey];
AVURLAsset *anAssetToUseInAComposition = [[AVURLAsset alloc]
initWithURL:url options:options];
```

Accessing the User's Assets

To access the assets managed the iPod Library or by the Photos application, you need to get a URL of the asset you want.

- To access the iPod Library, you create an `MPMediaQuery` instance to find the item you want, then get its URL using `MPMediaItemPropertyAssetURL`.

  For more about the Media Library, see *Multimedia Programming Guide*.
- To access the assets managed by the Photos application, you use `ALAssetsLibrary`.

The following example shows how you can get an asset to represent the first video in the Saved Photos Album.

```
ALAssetsLibrary *library = [[ALAssetsLibrary alloc] init];

// Enumerate just the photos and videos group by using ALAssetsGroupSavedPhotos.
[library enumerateGroupsWithTypes:ALAssetsGroupSavedPhotos usingBlock:^(ALAssetsGroup
 *group, BOOL *stop) {

// Within the group enumeration block, filter to enumerate just videos.
[group setAssetsFilter:[ALAssetsFilter allVideos]];

// For this example, we're only interested in the first item.
[group enumerateAssetsAtIndexes:[NSIndexSet indexSetWithIndex:0]
                        options:0
                     usingBlock:^(ALAsset *alAsset, NSUInteger index, BOOL
*innerStop) {
```

```
                        // The end of the enumeration is signaled by asset ==
nil.
                        if (alAsset) {
                            ALAssetRepresentation *representation = [alAsset
defaultRepresentation];
                            NSURL *url = [representation url];
                            AVAsset *avAsset = [AVURLAsset URLAssetWithURL:url
options:nil];
                            // Do something interesting with the AV asset.
                        }
                    }];
                }
                failureBlock: ^(NSError *error) {
                    // Typically you should handle an error more gracefully than
 this.
                    NSLog(@"No groups");
                }];

[library release];
```

Preparing an Asset for Use

Initializing an asset (or track) does not necessarily mean that all the information that you might want to retrieve for that item is immediately available. It may require some time to calculate even the duration of an item (an MP3 file, for example, may not contain summary information). Rather than blocking the current thread while a value is being calculated, you should use the `AVAsynchronousKeyValueLoading` protocol to ask for values and get an answer back later through a completion handler you define using a block. (`AVAsset` and `AVAssetTrack` conform to the `AVAsynchronousKeyValueLoading` protocol.)

You test whether a value is loaded for a property using `statusOfValueForKey:error:`. When an asset is first loaded, the value of most or all of its properties is `AVKeyValueStatusUnknown`. To load a value for one or more properties, you invoke `loadValuesAsynchronouslyForKeys:completionHandler:`. In the completion handler, you take whatever action is appropriate depending on the property's status. You should always be prepared for loading to not complete successfully, either because it failed for some reason such as a network-based URL being inaccessible, or because the load was canceled. .

```
NSURL *url = <#A URL that identifies an audiovisual asset such as a movie file#>;
```

```
AVURLAsset *anAsset = [[AVURLAsset alloc] initWithURL:url options:nil];
NSArray *keys = [NSArray arrayWithObject:@"duration"];


[asset loadValuesAsynchronouslyForKeys:keys completionHandler:^() {


    NSError *error = nil;
    AVKeyValueStatus tracksStatus = [asset statusOfValueForKey:@"duration"
error:&error];
    switch (tracksStatus) {
        case AVKeyValueStatusLoaded:
            [self updateUserInterfaceForDuration];
            break;
        case AVKeyValueStatusFailed:
            [self reportError:error forAsset:asset];
            break;
        case AVKeyValueStatusCancelled:
            // Do whatever is appropriate for cancelation.
            break;
   }
}];
```

If you want to prepare an asset for playback, you should load its `tracks` property. For more about playing assets, see "Playback" (page 19).

Getting Still Images From a Video

To get still images such as thumbnails from an asset for playback, you use an `AVAssetImageGenerator` object. You initialize an image generator with your asset. Initialization may succeed, though, even if the asset possesses no visual tracks at the time of initialization, so if necessary you should test whether the asset has any tracks with the visual characteristic using `tracksWithMediaCharacteristic:`.

```
AVAsset anAsset = <#Get an asset#>;
if ([anAsset tracksWithMediaCharacteristic:AVMediaTypeVideo]) {
    AVAssetImageGenerator *imageGenerator =
        [AVAssetImageGenerator assetImageGeneratorWithAsset:anAsset];
```

```
    // Implementation continues...
```

You can configure several aspects of the image generator, for example, you can specify the maximum dimensions for the images it generates and the aperture mode using `maximumSize` and `apertureMode` respectively.You can then generate a single image at a given time, or a series of images. You must ensure that you retain the image generator until it has generated all the images.

Generating a Single Image

You use `copyCGImageAtTime:actualTime:error:` to generate a single image at a specific time. AV Foundation may not be able to produce an image at exactly the time you request, so you can pass as the second argument a pointer to a CMTime that upon return contains the time at which the image was actually generated.

```
AVAsset *myAsset = <#An asset#>];
AVAssetImageGenerator *imageGenerator = [[AVAssetImageGenerator alloc]
initWithAsset:myAsset];


Float64 durationSeconds = CMTimeGetSeconds([myAsset duration]);
CMTime midpoint = CMTimeMakeWithSeconds(durationSeconds/2.0, 600);
NSError *error = nil;
CMTime actualTime;


CGImageRef halfWayImage = [imageGenerator copyCGImageAtTime:midpoint
actualTime:&actualTime error:&error];


if (halfWayImage != NULL) {


   NSString *actualTimeString = (NSString *)CMTimeCopyDescription(NULL, actualTime);
    NSString *requestedTimeString = (NSString *)CMTimeCopyDescription(NULL,
midpoint);
    NSLog(@"got halfWayImage: Asked for %@, got %@", requestedTimeString,
actualTimeString);
    [actualTimeString release];
    [requestedTimeString release];


    // Do something interesting with the image.
    CGImageRelease(halfWayImage);
```

```
}
[imageGenerator release];
```

Generating a Sequence of Images

To generate a series of images, you send the image generator a `generateCGImagesAsynchronouslyForTimes:completionHandler:` message. The first argument is an array of `NSValue` objects, each containing a CMTime, specifying the asset times for which you want images to be generated. The second argument is a block that serves as a callback invoked for each image that is generated. The block arguments provide a result constant that tells you whether the image was created successfully or if the operation was canceled, and, as appropriate:

- The image.
- The time for which you requested the image and the actual time for which the image was generated.
- An error object that describes the reason generation failed.

In your implementation of the block, you should check the result constant to determine whether the image was created. In addition, you must ensure that you retain the image generator until it has finished creating the images.

```
AVAsset *myAsset = <#An asset#>];
// Assume: @property (retain) AVAssetImageGenerator *imageGenerator;
self.imageGenerator = [AVAssetImageGenerator assetImageGeneratorWithAsset:myAsset];

Float64 durationSeconds = CMTimeGetSeconds([myAsset duration]);
CMTime firstThird = CMTimeMakeWithSeconds(durationSeconds/3.0, 600);
CMTime secondThird = CMTimeMakeWithSeconds(durationSeconds*2.0/3.0, 600);
CMTime end = CMTimeMakeWithSeconds(durationSeconds, 600);
NSArray *times = [NSArray arrayWithObjects:[NSValue valueWithCMTime:kCMTimeZero],
                  [NSValue valueWithCMTime:firstThird], [NSValue
valueWithCMTime:secondThird],
                  [NSValue valueWithCMTime:end], nil];

[imageGenerator generateCGImagesAsynchronouslyForTimes:times
               completionHandler:^(CMTime requestedTime, CGImageRef image, CMTime
 actualTime,
```

```
                                        AVAssetImageGeneratorResult result, NSError
*error) {


                NSString *requestedTimeString = (NSString
*)CMTimeCopyDescription(NULL, requestedTime);
              NSString *actualTimeString = (NSString *)CMTimeCopyDescription(NULL,
 actualTime);
                NSLog(@"Requested: %@; actual %@", requestedTimeString,
actualTimeString);
                [requestedTimeString release];
                [actualTimeString release];


                if (result == AVAssetImageGeneratorSucceeded) {
                    // Do something interesting with the image.
                }


                if (result == AVAssetImageGeneratorFailed) {
                   NSLog(@"Failed with error: %@", [error localizedDescription]);
                }
                if (result == AVAssetImageGeneratorCancelled) {
                    NSLog(@"Canceled");
                }
  }];
```

You can cancel the generation of the image sequence by sending the image generator a `cancelAllCGImageGeneration` message.

Trimming and Transcoding a Movie

You can transcode a movie from one format to another, and trim a movie, using an `AVAssetExportSession` object. An export session is a controller object that manages asynchronous export of an asset. You initialize the session using the asset you want to export and the name of a export preset that indicates the export

options you want to apply (see `allExportPresets`). You then configure the export session to specify the output URL and file type, and optionally other settings such as the metadata and whether the output should be optimized for network use.

You can check whether you can export a given asset using a given preset using `exportPresetsCompatibleWithAsset:` as illustrated in this example:

```
AVAsset *anAsset = <#Get an asset#>;
NSArray *compatiblePresets = [AVAssetExportSession
exportPresetsCompatibleWithAsset:anAsset];
if ([compatiblePresets containsObject:AVAssetExportPresetLowQuality]) {
    AVAssetExportSession *exportSession = [[AVAssetExportSession alloc]
        initWithAsset:anAsset presetName:AVAssetExportPresetLowQuality];
    // Implementation continues.
}
```

You complete configuration of the session by providing the output URL (The URL must be a file URL.) `AVAssetExportSession` can infer the output file type from the URL's path extension; typically, however, you set it directly using `outputFileType`. You can also specify additional properties such as the time range, a limit for the output file length, whether the exported file should be optimized for network use, and a video composition. The following example illustrates how to use the `timeRange` property to trim the movie:

```
    exportSession.outputURL = <#A file URL#>;
    exportSession.outputFileType = AVFileTypeQuickTimeMovie;

    CMTime start = CMTimeMakeWithSeconds(1.0, 600);
    CMTime duration = CMTimeMakeWithSeconds(3.0, 600);
    CMTimeRange range = CMTimeRangeMake(start, duration);
    exportSession.timeRange = range;
```

To create the new file you invoke `exportAsynchronouslyWithCompletionHandler:`. The completion handler block is called when the export operation finishes; in your implementation of the handler, you should check the session's `status` to determine whether the export was successful, failed, or was canceled:

```
    [exportSession exportAsynchronouslyWithCompletionHandler:^{


        switch ([exportSession status]) {
            case AVAssetExportSessionStatusFailed:
                NSLog(@"Export failed: %@", [[exportSession error]
localizedDescription]);
                break;
            case AVAssetExportSessionStatusCancelled:
                NSLog(@"Export canceled");
                break;
            default:
                break;
        }
        [exportSession release];
    }];
```

You can cancel the export by sending the session a `cancelExport` message.

The export will fail if you try to overwrite an existing file, or write a file outside of the application's sandbox. It may also fail if:

- There is an incoming phone call
- Your application is in the background and another application starts playback

In these situations, you should typically inform the user that the export failed, then allow the user to restart the export.

Reading and Writing Assets

On iOS 4.1 and later, you use an `AVAssetReader` when you want to perform an operation on the contents of an asset. For example, you might read the audio track of an asset to produce a visual representation of the waveform.

To produce an asset from media such as sample buffers or still images on iOS 4.1 and later, you use an `AVAssetWriter` object.

You can use an asset reader and asset writer object in tandem to convert an asset from one representation to another. Using these objects you have more control over the conversion than you do with `AVExportSession`. For example of you want to choose which of the tracks you want to be represented in the output file, specify your own output format, or modify the asset during the conversion process.

Playback

To control the playback of assets, you use an `AVPlayer` object. During playback, you can use an `AVPlayerItem` object to manage the presentation state of an asset as a whole, and an `AVPlayerItemTrack` to manage the presentation state of an individual track. To display video, you use an `AVPlayerLayer` object.

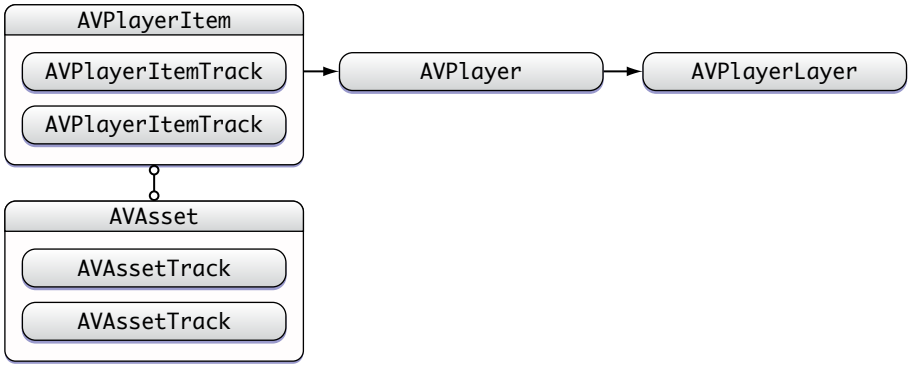
Playing Assets

A player is a controller object that you use to manage playback of an asset, for example starting and stopping playback, and seeking to a particular time. You use an instance of `AVPlayer` to play a single asset. On iOS 4.1 and later, you can use an `AVQueuePlayer` object to play a number of items in sequence (`AVQueuePlayer` is a subclass of `AVPlayer`).

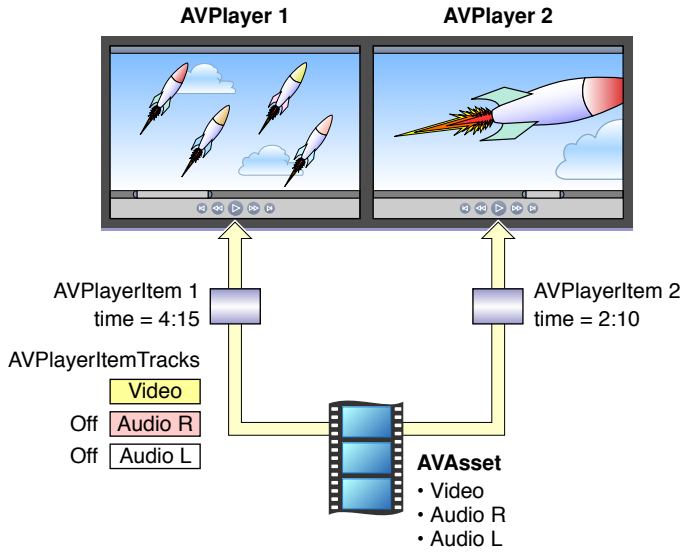
A player provides you with information about the state of the playback so, if you need to, you can synchronize your user interface with the player's state. You typically direct the output of a player to specialized Core Animation Layer (an instance of `AVPlayerLayer` or `AVSynchronizedLayer`). To learn more about layers, see *Core Animation Programming Guide* .

> **Multiple player layers** You can create arbitrarily many `AVPlayerLayer` objects from a single `AVPlayer` instance, but only the most-recently-created such layer will display any video content on-screen.

Although ultimately you want to play an asset, you don't provide assets directly to an `AVPlayer` object. Instead, you provide an instance of `AVPlayerItem`. A player item manages the presentation state of an asset with which it is associated. A player item contains player item tracks—instances of `AVPlayerItemTrack`—that correspond to the tracks in the asset.



This abstraction means that you can play a given asset using different players simultaneously, but rendered in different ways by each player. Using the item tracks, you can, for example, disable a particular track during playback (you might not want to play the sound component).



You can initialize a player item with an existing asset, or you can initialize a player item directly from a URL so that you can play a resource at a particular location (`AVPlayerItem` will then create and configure an asset for the resource). As with `AVAsset`, though, simply initializing a player item doesn't necessarily mean it's ready for immediate playback. You can observe (using key-value observing) an item's `status` property to determine if and when it's ready to play.

Handling Different Types of Asset

The way you configure an asset for playback may depend on the sort of asset you want to play. Broadly speaking, there are two main types: file-based assets, to which you have random access (such as from a local file, the camera roll, or the Media Library), and stream-based (HTTP Live Stream format).

To load and play a file-based asset. There are several steps to playing a file-based asset:

- Create an asset using `AVURLAsset` and load its tracks using `loadValuesAsynchronouslyForKeys:completionHandler:`.
- When the asset has loaded its tracks, create an instance of `AVPlayerItem` using the asset.
- Associate the item with an instance of `AVPlayer`.
- Wait until the item's `status` indicates that it's ready to play (typically you use key-value observing to receive a notification when the status changes).

This approach is illustrated in "Putting it all Together: Playing a Video File Using AVPlayerLayer" (page 27).

To create and prepare an HTTP live stream for playback. Initialize an instance of `AVPlayerItem` using the URL. (You cannot directly create an `AVAsset` instance to represent the media in an HTTP Live Stream.)

```
NSURL *url = [NSURL URLWithString:@"<#Live stream URL#>];

// You may find a test stream at
<http://devimages.apple.com/iphone/samples/bipbop/bipbopall.m3u8>.

self.playerItem = [AVPlayerItem playerItemWithURL:url];

[playerItem addObserver:self forKeyPath:@"status" options:0
context:&ItemStatusContext];

self.player = [AVPlayer playerWithPlayerItem:playerItem];
```

When you associate the player item with a player, it starts to become ready to play. When it is ready to play, the player item creates the `AVAsset` and `AVAssetTrack` instances, which you can use to inspect the contents of the live stream.

If you simply want to play a live stream, you can take a shortcut and create a player directly using the URL:

```
self.player = [AVPlayer playerWithURL:<#Live stream URL#>];
[player addObserver:self forKeyPath:@"status" options:0
context:&PlayerStatusContext];
```

As with assets and items, initializing the player does not mean it's ready for playback. You should observe the player's `status` property, which changes to `AVPlayerStatusReadyToPlay` when it is ready to play. You can also observe the `currentItem` property to access the player item created for the stream.

If you don't know what kind of URL you have. Follow these steps:

1. Try to initialize an `AVURLAsset` using the URL, then load its `tracks` key.

   If the tracks load successfully, then you create a player item for the asset.

2. If 1 fails, create an `AVPlayerItem` directly from the URL.

   Observe the player's `status` property to determine whether it becomes playable.

If either route succeeds, you end up with a player item that you can then associate with a player.

Playing an Item

To start playback, you send a `play` message to the player.

```
- (IBAction)play:sender {
    [player play];
}
```

In addition to simply playing, you can manage various aspects of the playback, such as the rate and the location of the playhead. You can also monitor the play state of the player; this is useful if you want to, for example, synchronize the user interface to the presentation state of the asset—see "Monitoring Playback" (page 24).

Changing the Playback Rate

You change the rate of playback by setting the player's `rate` property.

```
aPlayer.rate = 0.5;
aPlayer.rate = 2.0;
```

A value of 1.0 means "play at the natural rate of the current item." Setting the rate to 0.0 is the same as pausing playback—you can also use `pause`.

Seeking—Repositioning the Playhead

To move the playhead to a particular time, you generally use `seekToTime:`.

```
CMTime fiveSecondsIn = CMTimeMake(5, 1);
[player seekToTime:fiveSecondsIn];
```

The `seekToTime:` method, however, is tuned for performance rather than precision. If you need to move the playhead precisely, instead you use `seekToTime:toleranceBefore:toleranceAfter:`.

```
CMTime fiveSecondsIn = CMTimeMake(5, 1);
[player seekToTime:fiveSecondsIn toleranceBefore:kCMTimeZero
toleranceAfter:kCMTimeZero];
```

Using a tolerance of zero may require the framework to decode a large amount of data. You should only use zero if you are, for example, writing a sophisticated media editing application that requires precise control.

After playback, the player's head is set to the end of the item, and further invocations of `play` have no effect. To position the play head back at the beginning of the item, you can register to receive an `AVPlayerItemDidPlayToEndTimeNotification` from the item. In the notification's callback method, you invoke `seekToTime:` with the argument `kCMTimeZero`.

```
// Register with the notification center after creating the player item.
    [[NSNotificationCenter defaultCenter]
        addObserver:self
        selector:@selector(playerItemDidReachEnd:)
        name:AVPlayerItemDidPlayToEndTimeNotification
        object:<#The player item#>];

- (void)playerItemDidReachEnd:(NSNotification *)notification {
    [player seekToTime:kCMTimeZero];
}
```

Playing Multiple Items

On iOS 4.1 and later, you can use an `AVQueuePlayer` object to play a number of items in sequence. `AVQueuePlayer` is a subclass of `AVPlayer`. You initialize a queue player with an array of player items:

```
NSArray *items = <#An array of player items#>;
AVQueuePlayer *queuePlayer = [[AVQueuePlayer alloc] initWithItems:items];
```

You can then play the queue using `play`, just as you would an `AVPlayer` object. The queue player plays each item in turn. If you want to skip to the next item, you send the queue player an `advanceToNextItem` message.

You can modify the queue using `insertItem:afterItem:`, `removeItem:`, and `removeAllItems`. When adding a new item, you should typically check whether it can be inserted into the queue, using `canInsertItem:afterItem:`. You pass `nil` as the second argument to test whether the new item can be appended to the queue:

```
AVPlayerItem *anItem = <#Get a player item#>;
if ([queuePlayer canInsertItem:anItem afterItem:nil]) {
    [queuePlayer insertItem:anItem afterItem:nil];
}
```

Monitoring Playback

You can monitor a number of aspects of the presentation state of a player and the player item being played. This is particularly useful for state changes that are not under your direct control, for example:

- If the user uses multitasking to switch to a different application, a player's `rate` property will drop to `0.0`.
- If you are playing remote media, a player item's `loadedTimeRanges` and `seekableTimeRanges` properties will change as more data becomes available.

  These properties tell you what portions of the player item's timeline are available.
- A player's `currentItem` property changes as a player item is created for an HTTP live stream.
- A player item's `tracks` property may change while playing an HTTP live stream.

  This may happen if the stream offers different encodings for the content; the tracks change if the player switches to a different encoding.
- A player or player item's `status` may change if playback fails for some reason.

You can use key-value observing to monitor changes to values of these properties.

Important You should register for KVO change notifications and unregister from KVO change notifications on the main thread. This avoids the possibility of receiving a partial notification if a change is being made on another thread. AV Foundation invokes `observeValueForKeyPath:ofObject:change:context:` on the main thread, even if the change operation is made on another thread.

Responding to a Change in Status

When a player or player item's status changes, it emits a key-value observing change notification. If an object is unable to play for some reason (for example, if the media services are reset), the status changes to `AVPlayerStatusFailed` or `AVPlayerItemStatusFailed` as appropriate. In this situation, the value of the object's `error` property is changed to an error object that describes why the object is no longer be able to play.

AV Foundation does not specify what thread that the notification is sent on. If you want to update the user interface, you must make sure that any relevant code is invoked on the main thread. This example uses `dispatch_async(3) Mac OS X Manual Page` to execute code on the main thread.

```
- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object
                        change:(NSDictionary *)change context:(void *)context {

    if (context == <#Player status context#>) {
        AVPlayer *thePlayer = (AVPlayer *)object;
        if ([thePlayer status] == AVPlayerStatusFailed) {
            NSError *error = [<#The AVPlayer object#> error];
            // Respond to error: for example, display an alert sheet.
            return;
        }
        // Deal with other status change if appropriate.
    }
    // Deal with other change notifications if appropriate.
    [super observeValueForKeyPath:keyPath ofObject:object
           change:change context:context];
    return;
}
```

Tracking Readiness for Visual Display

You can observe an `AVPlayerLayer` object's `readyForDisplay` property to be notified when the layer has user-visible content. In particular, you might insert the player layer into the layer tree only when there is something for the user to look at, and perform a transition from

Tracking Time

To track changes in the position of the playhead in an `AVPlayer` object, you can use `addPeriodicTimeObserverForInterval:queue:usingBlock:` or `addBoundaryTimeObserverForTimes:queue:usingBlock:`. You might do this to, for example, update your user interface with information about time elapsed or time remaining, or perform some other user interface synchronization.

- With `addPeriodicTimeObserverForInterval:queue:usingBlock:`, the block you provide is invoked at the interval you specify, and if time jumps, and when playback starts or stops.
- With `addBoundaryTimeObserverForTimes:queue:usingBlock:`, you pass an array of CMTimes contained in `NSValue` objects. The block you provide is invoked whenever any of those times is traversed.

Both of the methods return an opaque object that serves as an observer. You must retain the returned object as long as you want the time observation block to be invoked by the player. You must also balance each invocation of these methods with a corresponding call to `removeTimeObserver:`.

With both of these methods, AV Foundation does not guarantee to invoke your block for every interval or boundary passed. AV Foundation does not invoke a block if execution of a previously-invoked block has not completed. You must make sure, therefore, that the work you perform in the block does not overly tax the system.

```
// Assume a property: @property (retain) id playerObserver;


Float64 durationSeconds = CMTimeGetSeconds([<#An asset#> duration]);
CMTime firstThird = CMTimeMakeWithSeconds(durationSeconds/3.0, 1);
CMTime secondThird = CMTimeMakeWithSeconds(durationSeconds*2.0/3.0, 1);
NSArray *times = [NSArray arrayWithObjects:[NSValue valueWithCMTime:firstThird],
[NSValue valueWithCMTime:secondThird], nil];


self.playerObserver = [<#A player#> addBoundaryTimeObserverForTimes:times queue:NULL
 usingBlock:^{
```

```
    NSString *timeDescription = (NSString *)CMTimeCopyDescription(NULL, [self.player
 currentTime]);
    NSLog(@"Passed a boundary at %@", timeDescription);
    [timeDescription release];
}];
```

Reaching the End of an Item

You can register to receive an `AVPlayerItemDidPlayToEndTimeNotification` notification when a player item has completed playback:

```
[[NSNotificationCenter defaultCenter] addObserver:<#The observer, typically self#>
                                         selector:@selector(<#The selector name#>)

name:AVPlayerItemDidPlayToEndTimeNotification
                                           object:<#A player item#>];
```

Putting it all Together: Playing a Video File Using AVPlayerLayer

This brief code example to illustrates how you can use an `AVPlayer` object to play a video file. It shows how to:

- Configure a view to use an `AVPlayerLayer` layer
- Create an `AVPlayer` object
- Create an `AVPlayerItem` object for a file-based asset, and use key-value observing to observe its status
- Respond to the item becoming ready to play by enabling a button
- Play the item, then restore the player's head to the beginning.

> **Note** To focus on the most relevant code, this example omits several aspects of a complete application, such as memory management, and unregistering as an observer (for key-value observing or for the notification center). To use AV Foundation, you are expected to have enough experience with Cocoa to be able to infer the missing pieces.

For a conceptual introduction to playback, skip to "Playing Assets" (page 19).

The Player View

To play the visual component of an asset, you need a view containing an `AVPlayerLayer` layer to which the output of an `AVPlayer` object can be directed. You can create a simple subclass of `UIView` to accommodate this:

```
#import <UIKit/UIKit.h>
#import <AVFoundation/AVFoundation.h>
@interface PlayerView : UIView {
}
@property (nonatomic, retain) AVPlayer *player;
@end

@implementation PlayerView
+ (Class)layerClass {
    return [AVPlayerLayer class];
}
- (AVPlayer*)player {
    return [(AVPlayerLayer *)[self layer] player];
}
- (void)setPlayer:(AVPlayer *)player {
    [(AVPlayerLayer *)[self layer] setPlayer:player];
}
@end
```

A Simple View Controller

Assume you have a simple view controller, declared as follows:

```
@class PlayerView;
@interface PlayerViewController : UIViewController {
}
@property (nonatomic, retain) AVPlayer *player;
@property (retain) AVPlayerItem *playerItem;
@property (nonatomic, retain) IBOutlet PlayerView *playerView;
@property (nonatomic, retain) IBOutlet UIButton *playButton;
- (IBAction)loadAssetFromFile:sender;
```

```
- (IBAction)play:sender;
- (void)syncUI;
@end
```

The `syncUI` method synchronizes the button's state with the player's state:

```
- (void)syncUI {
    if ((player.currentItem != nil) &&
        ([player.currentItem status] == AVPlayerItemStatusReadyToPlay)) {
        playButton.enabled = YES;
    }
    else {
        playButton.enabled = NO;
    }
}
```

You can invoke `syncUI` in the view controller's `viewDidLoad` method to ensure a consistent user interface when the view is first displayed.

```
- (void)viewDidLoad {
    [super viewDidLoad];
    [self syncUI];
}
```

The other properties and methods are described in the remaining sections.

Creating the Asset

You create an asset from a URL using `AVURLAsset`. Creating the asset, however, does not necessarily mean that it's ready for use. To be used, an asset must have loaded its tracks. To avoid blocking the current thread, you load the asset's tracks asynchronously using `loadValuesAsynchronouslyForKeys:completionHandler:`. (The following example assumes your project contains a suitable video resource.)

```
- (IBAction)loadAssetFromFile:sender {
```

```
    NSURL *fileURL = [[NSBundle mainBundle]
        URLForResource:<#@"VideoFileName"#> withExtension:<#@"extension"#>];

    AVURLAsset *asset = [AVURLAsset URLAssetWithURL:fileURL options:nil];
    NSString *tracksKey = @"tracks";

    [asset loadValuesAsynchronouslyForKeys:[NSArray arrayWithObject:tracksKey]
completionHandler:
     ^{
         // The completion block goes here.
     }];
}
```

In the completion block, you create an instance of `AVPlayerItem` for the asset, and set it as the player for the player view. As with creating the asset, simply creating the player item does not mean it's ready to use. To determine when it's ready to play, you can observe the item's `status`. You trigger its preparation to play when you associate it with the player.

```
// Define this constant for the key-value observation context.
static const NSString *ItemStatusContext;

// Completion handler block.
         dispatch_async(dispatch_get_main_queue(),
            ^{
                NSError *error = nil;
                AVKeyValueStatus status = [asset statusOfValueForKey:tracksKey
error:&error];

                if (status == AVKeyValueStatusLoaded) {
                    self.playerItem = [AVPlayerItem playerItemWithAsset:asset];
                    [playerItem addObserver:self forKeyPath:@"status"
                                options:0 context:&ItemStatusContext];
                    [[NSNotificationCenter defaultCenter] addObserver:self

selector:@selector(playerItemDidReachEnd:)
```

```
name:AVPlayerItemDidPlayToEndTimeNotification
                                                                object:playerItem];
                    self.player = [AVPlayer playerWithPlayerItem:playerItem];
                    [playerView setPlayer:player];
                }
                else {
                    // You should deal with the error appropriately.
                    NSLog(@"The asset's tracks were not loaded:\n%@", [error
localizedDescription]);
                }
            });
```

Responding to the Player Item's Status Change

When the player item's status changes, the view controller receives a key-value observing change notification. AV Foundation does not specify what thread that the notification is sent on. If you want to update the user interface, you must make sure that any relevant code is invoked on the main thread. This example uses `dispatch_async(3) Mac OS X Manual Page` to queue a message on the main thread to synchronize the user interface.

```
- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object
                        change:(NSDictionary *)change context:(void *)context {

    if (context == &ItemStatusContext) {
        dispatch_async(dispatch_get_main_queue(),
                       ^{
                           [self syncUI];
                       });
        return;
    }
    [super observeValueForKeyPath:keyPath ofObject:object
           change:change context:context];
    return;
}
```

Playing the Item

Playing the item is trivial: you send a `play` message to the player.

```
- (IBAction)play:sender {
    [player play];
}
```

This only plays the item once, though. After playback, the player's head is set to the end of the item, and further invocations of `play` will have no effect. To position the play head back at the beginning of the item, you can register to receive an `AVPlayerItemDidPlayToEndTimeNotification` from the item. In the notification's callback method, invoke `seekToTime:` with the argument `kCMTimeZero`.

```
// Register with the notification center after creating the player item.
    [[NSNotificationCenter defaultCenter]
        addObserver:self
        selector:@selector(playerItemDidReachEnd:)
        name:AVPlayerItemDidPlayToEndTimeNotification
        object:[player currentItem]];

- (void)playerItemDidReachEnd:(NSNotification *)notification {
    [player seekToTime:kCMTimeZero];
}
```

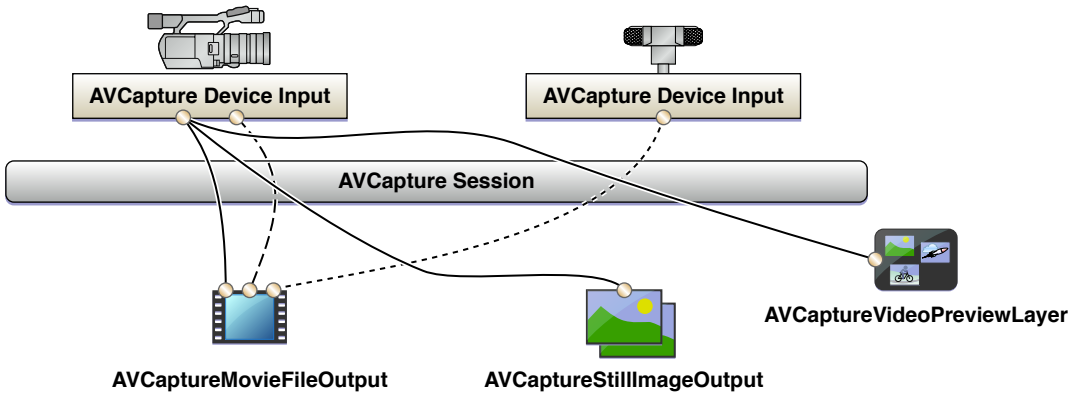
Media Capture

To manage the capture from a device such as a camera or microphone, you assemble objects to represent inputs and outputs, and use an instance of `AVCaptureSession` to coordinate the data flow between them. Minimally you need:

- An instance of `AVCaptureDevice` to represent the input device, such as a camera or microphone
- An instance of a concrete subclass of `AVCaptureInput` to configure the ports from the input device
- An instance of a concrete subclass of `AVCaptureOutput` to manage the output to a movie file or still image
- An instance of `AVCaptureSession` to coordinate the data flow from the input to the output

To show the user what a camera is recording, you can use an instance of `AVCaptureVideoPreviewLayer` (a subclass of `CALayer`).

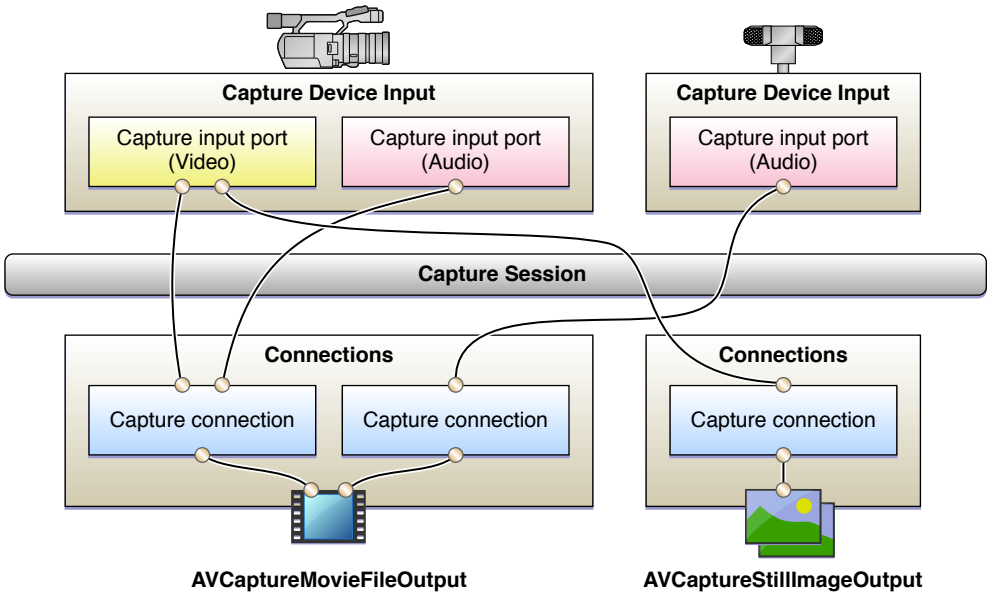
You can configure multiple inputs and outputs, coordinated by a single session:



For many applications, this is as much detail as you need. For some operations, however, (if you want to monitor the power levels in an audio channel, for example) you need to consider how the various ports of an input device are represented, how those ports are connected to the output.

A connection between a capture input and a capture output in a capture session is represented by an `AVCaptureConnection` object. Capture inputs (instances of `AVCaptureInput`) have one or more input ports (instances of `AVCaptureInputPort`). Capture outputs (instances of `AVCaptureOutput`) can accept data from one or more sources (for example, an `AVCaptureMovieFileOutput` object accepts both video and audio data).

When you add an input or an output to a session, the session "greedily" forms connections between all the compatible capture inputs' ports and capture outputs. A connection between a capture input and a capture output is represented by an `AVCaptureConnection` object.



You can use a capture connection to enable or disable the flow of data from a given input or to a given output. You can also use a connection to monitor the average and peak power levels in an audio channel.

Use a Capture Session to Coordinate Data Flow

`AVCaptureSession` object is the central coordinating object you use to manage data capture. You use an instance to coordinate the flow of data from AV input devices to outputs. You add the capture devices and outputs you want to the session, then start data flow by sending the session a `startRunning` message, and stop recording by sending a `stopRunning` message.

```
AVCaptureSession *session = [[AVCaptureSession alloc] init];
// Add inputs and outputs.
[session startRunning];
```

Configuring a Session

You use a **preset** on the session to specify the image quality and resolution you want. A preset is a constant that identifies one of a number of possible configurations; in some cases the actual configuration is device-specific:

Symbol	Resolution	Comments
`AVCaptureSessionPresetHigh`	High	Highest recording quality. This varies per device.
`AVCaptureSessionPresetMedium`	Medium	Suitable for WiFi sharing. The actual values may change.
`AVCaptureSessionPresetLow`	Low	Suitable for 3G sharing. The actual values may change.
`AVCaptureSessionPreset640x480`	640x480	VGA.
`AVCaptureSessionPreset1280x720`	1280x720	720p HD.
`AVCaptureSessionPresetPhoto`	Photo	Full photo resolution. This is not supported for video output.

For examples of the actual values these presets represent for various devices, see "Saving to a Movie File" (page 44) and "Capturing Still Images" (page 48).

If you want to set a size-specific configuration, you should check whether it is supported before setting it:

```
if ([session canSetSessionPreset:AVCaptureSessionPreset1280x720]) {
    session.sessionPreset = AVCaptureSessionPreset1280x720;
}
else {
    // Handle the failure.
}
```

In many situations, you create a session and the various inputs and outputs all at once. Sometimes, however, you may want to reconfigure a running session, perhaps as different input devices become available, or in response to user request. This can present a challenge, since, if you change them one at a time, a new setting may be incompatible with an existing setting. To deal with this, you use `beginConfiguration` and `commitConfiguration` to batch multiple configuration operations into an atomic update. After calling

`beginConfiguration`, you can for example add or remove outputs, alter the `sessionPreset`, or configure individual capture input or output properties. No changes are actually made until you invoke `commitConfiguration`, at which time they are applied together.

```
[session beginConfiguration];
// Remove an existing capture device.
// Add a new capture device.
// Reset the preset.
[session commitConfiguration];
```

Monitoring Capture Session State

A capture session posts notifications that you can observe to be notified, for example, when it starts or stops running, or when it is interrupted. You can also register to receive an `AVCaptureSessionRuntimeErrorNotification` if a runtime error occurs. You can also interrogate the session's `running` property to find out if it is running, and its `interrupted` property to find out if it is interrupted.

An AVCaptureDevice Object Represents an Input Device

An `AVCaptureDevice` object abstracts a physical capture device that provides input data (such as audio or video) to an `AVCaptureSession` object. There is one object for each input device, so for example on an iPhone 3GS there is one video input for the camera and one audio input for the microphone; on an iPhone 4 there are two video inputs—one for front-facing the camera, one for the back-facing camera—and one audio input for the microphone.

You can find out what capture devices are currently available using the `AVCaptureDevice` class methods `devices` and `devicesWithMediaType:`, and if necessary find out what features the devices offer (see "Device Capture Settings" (page 37)). The list of available devices may change, though. Current devices may become unavailable (if they're used by another application), and new devices may become available, (if they're relinquished by another application). You should register to receive `AVCaptureDeviceWasConnectedNotification` and `AVCaptureDeviceWasDisconnectedNotification`notifications to be alerted when the list of available devices changes.

You add a device to a capture session using a capture input (see "Use Capture Inputs to Add a Capture Device to a Session" (page 42)).

Device Characteristics

You can ask a device about several different characteristics. You can test whether it provides a particular media type or supports a given capture session preset using `hasMediaType:` and `supportsAVCaptureSessionPreset:` respectively. To provide information to the user, you can find out the position of the capture device (whether it is on the front or the back of the unit they're using), and its localized name. This may be useful if you want to present a list of capture devices to allow the user to choose one.

The following code example iterates over all the available devices and logs their name, and for video devices their position on the unit.

```
NSArray *devices = [AVCaptureDevice devices];

for (AVCaptureDevice *device in devices) {

    NSLog(@"Device name: %@", [device localizedName]);

    if ([device hasMediaType:AVMediaTypeVideo]) {

        if ([device position] == AVCaptureDevicePositionBack) {
            NSLog(@"Device position : back");
        }
        else {
            NSLog(@"Device position : front");
        }
    }
}
```

In addition, you can find out the device's model ID and its unique ID.

Device Capture Settings

Different devices have different capabilities; for example, some may support different focus or flash modes; some may support focus on a point of interest.

Feature	iPhone 3G	iPhone 3GS	iPhone 4 (Back)	iPhone 4 (Front)
Focus mode	NO	YES	YES	NO

Feature	iPhone 3G	iPhone 3GS	iPhone 4 (Back)	iPhone 4 (Front)
Focus point of interest	NO	YES	YES	NO
Exposure mode	YES	YES	YES	YES
Exposure point of interest	NO	YES	YES	YES
White balance mode	YES	YES	YES	YES
Flash mode	NO	NO	YES	NO
Torch mode	NO	NO	YES	NO

The following code fragment shows how you can find video input devices that have a torch mode and support a given capture session preset:

```
NSArray *devices = [AVCaptureDevice devicesWithMediaType:AVMediaTypeVideo];
NSMutableArray *torchDevices = [[NSMutableArray alloc] init];

for (AVCaptureDevice *device in devices) {
    [if ([device hasTorch] &&
         [device supportsAVCaptureSessionPreset:AVCaptureSessionPreset640x480]) {
        [torchDevices addObject:device];
    }
}
```

If you find multiple devices that meet your criteria, you might let the user choose which one they want to use. To display a description of a device to the user, you can use its `localizedName` property.

You use the various different features in similar ways. There are constants to specify a particular mode, and you can ask a device whether it supports a particular mode. In several cases you can observe a property to be notified when a feature is changing. In all cases, you should lock the device before changing the mode of a particular feature, as described in "Configuring a Device" (page 41).

> **Note** Focus point of interest and exposure point of interest are mutually exclusive, as are focus mode and exposure mode.

Focus modes

There are three focus modes:

- `AVCaptureFocusModeLocked`: the focal length is fixed.

  This is useful when you want to allow the user to compose a scene then lock the focus.

- `AVCaptureFocusModeAutoFocus`: the camera does a single scan focus then reverts to locked.

  This is suitable for a situation where you want to select a particular item on which to focus and then maintain focus on that item even if it is not the center of the scene.

- `AVCaptureFocusModeContinuousAutoFocus`: the camera continuously auto-focuses as needed.

You use the `isFocusModeSupported:` method to determine whether a device supports a given focus mode, then set the mode using the `focusMode` property.

In addition, a device may support a focus point of interest. You test for support using `focusPointOfInterestSupported`. If it's supported, you set the focal point using `focusPointOfInterest`. You pass a `CGPoint` where `{0,0}` represents the top left of the picture area, and `{1,1}` represents the bottom right *in landscape mode with the home button on the right* —this applies even if the device is in portrait mode.

You can use the `adjustingFocus` property to determine whether a device is currently focusing. You can observe the property using key-value observing to be notified when a device starts and stops focusing.

If you change the focus mode settings, you can return them to the default configuration as follows:

```
if ([currentDevice isFocusModeSupported:AVCaptureFocusModeContinuousAutoFocus]) {
    CGPoint autofocusPoint = CGPointMake(0.5f, 0.5f);
    [currentDevice setFocusPointOfInterest:autofocusPoint];
    [currentDevice setFocusMode:AVCaptureFocusModeContinuousAutoFocus];
}
```

Exposure modes

There are two exposure modes:

- `AVCaptureExposureModeLocked`: the exposure mode is fixed.
- `AVCaptureExposureModeAutoExpose`: the camera continuously changes the exposure level as needed.

You use the `isExposureModeSupported:` method to determine whether a device supports a given exposure mode, then set the mode using the `exposureMode` property.

In addition, a device may support an exposure point of interest. You test for support using `exposurePointOfInterestSupported`. If it's supported, you set the exposure point using `exposurePointOfInterest`. You pass a `CGPoint` where `{0,0}` represents the top left of the picture area, and `{1,1}` represents the bottom right *in landscape mode with the home button on the right* —this applies even if the device is in portrait mode.

You can use the `adjustingExposure` property to determine whether a device is currently changing its exposure setting. You can observe the property using key-value observing to be notified when a device starts and stops changing its exposure setting.

If you change the exposure settings, you can return them to the default configuration as follows:

```
if ([currentDevice
isExposureModeSupported:AVCaptureExposureModeContinuousAutoExposure]) {
    CGPoint exposurePoint = CGPointMake(0.5f, 0.5f);
    [currentDevice setExposurePointOfInterest:exposurePoint];
    [currentDevice setExposureMode:AVCaptureExposureModeContinuousAutoExposure];
}
```

Flash modes

There are three flash modes:

- `AVCaptureFlashModeOff`: the flash will never fire.
- `AVCaptureFlashModeOn`: the flash will always fire.
- `AVCaptureFlashModeAuto`: the flash will fire if needed.

You use `hasFlash` to determine whether a device has a flash. You use the `isFlashModeSupported:` method to determine whether a device supports a given flash mode, then set the mode using the `flashMode` property.

Torch mode

Torch mode is where a camera uses the flash continuously at a low power to illuminate a video capture. There are three torch modes:

- `AVCaptureTorchModeOff`: the torch is always off.
- `AVCaptureTorchModeOn`: the torch is always on.

- `AVCaptureTorchModeAuto`: the torch is switched on and off as needed.

You use `hasTorch` to determine whether a device has a flash. You use the `isTorchModeSupported:` method to determine whether a device supports a given flash mode, then set the mode using the `torchMode` property.

For devices with a torch, the torch only turns on if the device is associated with a running capture session.

White balance

There are two white balance modes:

- `AVCaptureWhiteBalanceModeLocked`: the white balance mode is fixed.
- `AVCaptureWhiteBalanceModeContinuousAutoWhiteBalance`: the camera continuously changes the white balance as needed.

You use the `isWhiteBalanceModeSupported:` method to determine whether a device supports a given white balance mode, then set the mode using the `whiteBalanceMode` property.

You can use the `adjustingWhiteBalance` property to determine whether a device is currently changing its white balance setting. You can observe the property using key-value observing to be notified when a device starts and stops changing its white balance setting.

Configuring a Device

To set capture properties on a device, you must first acquire a lock on the device using `lockForConfiguration:`. This avoids making changes that may be incompatible with settings in other applications. The following code fragment illustrates how to approach changing the focus mode on a device by first determining whether the mode is supported, then attempting to lock the device for reconfiguration. The focus mode is changed only if the lock is obtained, and the lock is released immediately afterward.

```
if ([device isFocusModeSupported:AVCaptureFocusModeLocked]) {
    NSError *error = nil;
    if ([device lockForConfiguration:&error]) {
        device.focusMode = AVCaptureFocusModeLocked;
        [device unlockForConfiguration];
    }
    else {
        // Respond to the failure as appropriate.
```

You should only hold the device lock if you need settable device properties to remain unchanged. Holding the device lock unnecessarily may degrade capture quality in other applications sharing the device.

Switching Between Devices

Sometimes you may want to allow the user to switch between input devices—for example, on an iPhone 4 they could switch from using the front to the back camera. To avoid pauses or stuttering, you can reconfigure a session while it is running, however you should use `beginConfiguration` and `commitConfiguration` to bracket your configuration changes:

```
AVCaptureSession *session = <#A capture session#>;
[session beginConfiguration];

[session removeInput:frontFacingCameraDeviceInput];
[session addInput:backFacingCameraDeviceInput];

[session commitConfiguration];
```

When the outermost `commitConfiguration` is invoked, all the changes are made together. This ensures a smooth transition.

Use Capture Inputs to Add a Capture Device to a Session

To add a capture device to a capture session, you use an instance of `AVCaptureDeviceInput` (a concrete subclass of the abstract `AVCaptureInput` class). The capture device input manages the device's ports.

```
NSError *error = nil;
AVCaptureDeviceInput *input =
        [AVCaptureDeviceInput deviceInputWithDevice:device error:&error];
if (!input) {
    // Handle the error appropriately.
}
```

You add inputs to a session using `addInput:`. If appropriate, you can check whether a capture input is compatible with an existing session using `canAddInput:`.

```
AVCaptureSession *captureSession = <#Get a capture session#>;
AVCaptureDeviceInput *captureDeviceInput = <#Get a capture device input#>;
if ([captureSession canAddInput:captureDeviceInput]) {
    [captureSession addInput:captureDeviceInput];
}
else {
    // Handle the failure.
}
```

See "Configuring a Session" (page 35) for more details on how you might reconfigure a running session.

An `AVCaptureInput` vends one or more streams of media data. For example, input devices can provide both audio and video data. Each media stream provided by an input is represented by an `AVCaptureInputPort` object. A capture session uses an `AVCaptureConnection` object to define the mapping between a set of `AVCaptureInputPort` objects and a single `AVCaptureOutput`.

Use Capture Outputs to Get Output from a Session

To get output from a capture session, you add one or more outputs. An output is an instance of a concrete subclass of `AVCaptureOutput`; you use:

- `AVCaptureMovieFileOutput` to output to a movie file
- `AVCaptureVideoDataOutput` if you want to process frames from the video being captured
- `AVCaptureAudioDataOutput` if you want to process the audio data being captured
- `AVCaptureStillImageOutput` if you want to capture still images with accompanying metadata

You add outputs to a capture session using `addOutput:`. You check whether a capture output is compatible with an existing session using `canAddOutput:`. You can add and remove outputs as you want while the session is running.

```
AVCaptureSession *captureSession = <#Get a capture session#>;
AVCaptureMovieFileOutput *movieInput = <#Create and configure a movie output#>;
if ([captureSession canAddOutput:movieInput]) {
    [captureSession addOutput:movieInput];
}
else {
```

```
    // Handle the failure.
}
```

Saving to a Movie File

You save movie data to a file using an `AVCaptureMovieFileOutput` object. (`AVCaptureMovieFileOutput` is a concrete subclass of `AVCaptureFileOutput`, which defines much of the basic behavior.) You can configure various aspects of the movie file output, such as the maximum duration of the recording, or the maximum file size. You can also prohibit recording if there is less than a given amount of disk space left.

```
AVCaptureMovieFileOutput *aMovieFileOutput = [[AVCaptureMovieFileOutput alloc]
init];

CMTime maxDuration = <#Create a CMTime to represent the maximum duration#>;

aMovieFileOutput.maxRecordedDuration = maxDuration;

aMovieFileOutput.minFreeDiskSpaceLimit = <#An appropriate minimum given the quality
 of the movie format and the duration#>;
```

The resolution and bit rate for the output depend on the capture session's `sessionPreset`. The video encoding is typically H.264 and audio encoding AAC. The actual values vary by device, as illustrated in the following table.

Preset	iPhone 3G	iPhone 3GS	iPhone 4 (Back)	iPhone 4 (Front)
High	No video Apple Lossless	640x480 3.5 mbps	1280x720 10.5 mbps	640x480 3.5 mbps
Medium	No video Apple Lossless	480x360 700 kbps	480x360 700 kbps	480x360 700 kbps
Low	No video Apple Lossless	192x144 128 kbps	192x144 128 kbps	192x144 128 kbps
640x480	No video Apple Lossless	640x480 3.5 mbps	640x480 3.5 mbps	640x480 3.5 mbps
1280x720	No video Apple Lossless	No video 64 kbps AAC	No video 64 kbps AAC	No video 64 kbps AAC
Photo	Not supported for video output	Not supported for video output	Not supported for video output	Not supported for video output

Starting a Recording

You start recording a QuickTime movie using `startRecordingToOutputFileURL:recordingDelegate:`. You need to supply a file-based URL and a delegate. The URL must not identify an existing file, as the movie file output does not overwrite existing resources. You must also have permission to write to the specified location. The delegate must conform to the `AVCaptureFileOutputRecordingDelegate` protocol, and must implement the `captureOutput:didFinishRecordingToOutputFileAtURL:fromConnections:error:` method.

```
AVCaptureMovieFileOutput *aMovieFileOutput = <#Get a movie file output#>;
NSURL *fileURL = <#A file URL that identifies the output location#>;
[aMovieFileOutput startRecordingToOutputFileURL:fileURL recordingDelegate:<#The
delegate#>];
```

In the implementation of `captureOutput:didFinishRecordingToOutputFileAtURL:fromConnections:error:`, the delegate might write the resulting movie to the camera roll. It should also check for any errors that might have occurred.

Ensuring the File Was Written Successfully

To determine whether the file was saved successfully, in the implementation of `captureOutput:didFinishRecordingToOutputFileAtURL:fromConnections:error:` you check not only the error, but also the value of the `AVErrorRecordingSuccessfullyFinishedKey` in the error's user info dictionary:

```
- (void)captureOutput:(AVCaptureFileOutput *)captureOutput
        didFinishRecordingToOutputFileAtURL:(NSURL *)outputFileURL
        fromConnections:(NSArray *)connections
        error:(NSError *)error {

    BOOL recordedSuccessfully = YES;
    if ([error code] != noErr) {
        // A problem occurred: Find out if the recording was successful.
        id value = [[error userInfo]
objectForKey:AVErrorRecordingSuccessfullyFinishedKey];
        if (value) {
            recordedSuccessfully = [value boolValue];
        }
    }
```

```
    // Continue as appropriate...
```

You should check the value of the `AVErrorRecordingSuccessfullyFinishedKey` in the error's user info dictionary because the file might have been saved successfully, even though you got an error. The error might indicate that one of your recording constraints was reached, for example `AVErrorMaximumDurationReached` or `AVErrorMaximumFileSizeReached`. Other reasons the recording might stop are:

- The disk is full—`AVErrorDiskFull`.
- The recording device was disconnected (for example, the microphone was removed from an iPod touch)—`AVErrorDeviceWasDisconnected`.
- The session was interrupted (for example, a phone call was received)—`AVErrorSessionWasInterrupted`.

Adding Metadata to a File

You can set metadata for the movie file at any time, even while recording. This is useful for situations where the information is not available when the recording starts, as may be the case with location information. Metadata for a file output is represented by an array of `AVMetadataItem` objects; you use an instance of its mutable subclass, `AVMutableMetadataItem`, to create metadata of your own.

```
AVCaptureMovieFileOutput *aMovieFileOutput = <#Get a movie file output#>;
NSArray *existingMetadataArray = aMovieFileOutput.metadata;
AVMutableArray *newMetadataArray = nil;
if (existingMetadataArray) {
    newMetadataArray = [existingMetadataArray mutableCopy];
}
else {
    newMetadataArray = [[AVMutableArray alloc] init];
}

AVMutableMetadataItem *item = [[AVMutableMetadataItem alloc] init];
item.keySpace = AVMetadataKeySpaceCommon;
item.key = AVMetadataCommonKeyLocation;

CLLocation *location – <#The location to set#>;
item.value = [NSString stringWithFormat:@"%+08.4lf%+09.4lf/"
    location.coordinate.latitude, location.coordinate.longitude];
```

```
[newMetadataArray addObject:item];

aMovieFileOutput.metadata = newMetadataArray;
```

Processing Frames of Video

An `AVCaptureVideoDataOutput` object uses delegation to vend video frames. You set the delegate using `setSampleBufferDelegate:queue:`. In addition to the delegate, you specify a serial queue on which they delegate methods are invoked. You must use a serial queue to ensure that frames are delivered to the delegate in the proper order. You should not pass the queue returned by `dispatch_get_current_queue` since there is no guarantee as to which thread the current queue is running on. You can use the queue to modify the priority given to delivering and processing the video frames.

The frames are presented in the delegate method, `captureOutput:didOutputSampleBuffer:fromConnection:`, as instances of the CMSampleBuffer opaque type (see "Representations of Media" (page 59)). By default, the buffers are emitted in the camera's most efficient format. You can use the `videoSettings` property to specify a custom output format. The video settings property is a dictionary; currently, the only supported key is `kCVPixelBufferPixelFormatTypeKey`. The recommended pixel format choices for iPhone 4 are `kCVPixelFormatType_420YpCbCr8BiPlanarVideoRange` or `kCVPixelFormatType_32BGRA`; for iPhone 3G the recommended pixel format choices are `kCVPixelFormatType_422YpCbCr8` or `kCVPixelFormatType_32BGRA`. Both Core Graphics and OpenGL work well with the BGRA format:

```
AVCaptureSession *captureSession = <#Get a capture session#>;

NSDictionary *newSettings = [NSDictionary
dictionaryWithObject:(id)kCVPixelFormatType_32BGRA

forKey:(id)kCVPixelBufferPixelFormatTypeKey];

captureSession.videoSettings = newSettings;
```

Performance Considerations for Processing Video

You should set the session output to the lowest practical resolution for your application. Setting the output to a higher resolution than necessary wastes processing cycles and needlessly consumes power.

You must ensure that your implementation of `captureOutput:didOutputSampleBuffer:fromConnection:` is able to process a sample buffer within the amount of time allotted to a frame. If it takes too long, and you hold onto the video frames, AV Foundation will stop delivering frames, not only to your delegate but also other outputs such as a preview layer.

You can use the capture video data output's `minFrameDuration` property to ensure you have enough time to process a frame—at the cost of having a lower frame rate than would otherwise be the case. You might also ensure that the `alwaysDiscardsLateVideoFrames` property is set to `YES` (the default). This ensures that any late video frames are dropped rather than handed to you for processing. Alternatively, if you are recording and it doesn't matter if the output fames are a little late, you would *prefer* to get all of them, you can set the property value to `NO`. This does not mean that frames will not be dropped (that is, frames may still be dropped), but they may not be dropped as early, or as efficiently.

Capturing Still Images

You use an `AVCaptureStillImageOutput` output if you want to capture still images with accompanying metadata. The resolution of the image depends on the preset for the session, as illustrated in this table:

Preset	iPhone 3G	iPhone 3GS	iPhone 4 (Back)	iPhone 4 (Front)
High	400x304	640x480	1280x720	640x480
Medium	400x304	480x360	480x360	480x360
Low	400x304	192x144	192x144	192x144
640x480	N/A	640x480	640x480	640x480
1280x720	N/A	N/A	1280x720	N/A
Photo	1600x1200	2048x1536	2592x1936	640x480

Pixel and Encoding Formats

Different devices support different image formats:

iPhone 3G	iPhone 3GS	iPhone 4
yuvs, 2vuy, BGRA, jpeg	420f, 420v, BGRA, jpeg	420f, 420v, BGRA, jpeg

You can find out what pixel and codec types are supported using `availableImageDataCVPixelFormatTypes` and `availableImageDataCodecTypes` respectively. You set the `outputSettings` dictionary to specify the image format you want, for example:

```
AVCaptureStillImageOutput *stillImageOutput = [[AVCaptureStillImageOutput alloc]
init];
NSDictionary *outputSettings = [[NSDictionary alloc] initWithObjectsAndKeys:
```

```
                                                    AVVideoCodecJPEG,
AVVideoCodecKey, nil];
[stillImageOutput setOutputSettings:outputSettings];
```

If you want to capture a JPEG image, you should typically not specify your own compression format. Instead, you should let the still image output do the compression for you, since its compression is hardware-accelerated. If you need a data representation of the image, you can use `jpegStillImageNSDataRepresentation:` to get an `NSData` object without re-compressing the data, even if you modify the image's metadata.

Capturing an Image

When you want to capture an image, you send the output a `captureStillImageAsynchronouslyFromConnection:completionHandler:` message. The first argument is the connection you want to use for the capture. You need to look for the connection whose input port is collecting video:

```
AVCaptureConnection *videoConnection = nil;
for (AVCaptureConnection *connection in stillImageOutput.connections) {
    for (AVCaptureInputPort *port in [connection inputPorts]) {
        if ([[port mediaType] isEqual:AVMediaTypeVideo] ) {
            videoConnection = connection;
            break;
        }
    }
    if (videoConnection) { break; }
}
```

The second argument to `captureStillImageAsynchronouslyFromConnection:completionHandler:` is a block that takes two arguments: a CMSampleBuffer containing the image data, and an error. The sample buffer itself may contain metadata, such as an Exif dictionary, as an attachment. You can modify the attachments should you want, but note the optimization for JPEG images discussed in "Pixel and Encoding Formats" (page 48).

```
[stillImageOutput captureStillImageAsynchronouslyFromConnection:videoConnection
completionHandler:
    ^(CMSampleBufferRef imageSampleBuffer, NSError *error) {
        CFDictionaryRef exifAttachments =
```

```
            CMGetAttachment(imageSampleBuffer, kCGImagePropertyExifDictionary,
NULL);
        if (exifAttachments) {
            // Do something with the attachments.
        }
        // Continue as appropriate.
    }];
```

Showing the User What's Being Recorded

You can provide the user with a preview of what's being recorded by the camera using a preview layer, or by the microphone by monitoring the audio channel.

Video Preview

You can provide the user with a preview of what's being recorded using an `AVCaptureVideoPreviewLayer` object. `AVCaptureVideoPreviewLayer` is a subclass of`CALayer` (see *Core Animation Programming Guide*. You don't need any outputs to show the preview.

Unlike a capture output, a video preview layer retains the session with which it is associated. This is to ensure that the session is not deallocated while the layer is attempting to display video. This is reflected in the way you initialize a preview layer:

```
AVCaptureSession *captureSession = <#Get a capture session#>;
CALayer *viewLayer = <#Get a layer from the view in which you want to present the
 preview#>;


AVCaptureVideoPreviewLayer *captureVideoPreviewLayer = [[AVCaptureVideoPreviewLayer
 alloc] initWithSession:captureSession];
[viewLayer addSublayer:captureVideoPreviewLayer];
```

In general, the preview layer behaves like any other `CALayer` object in the render tree (see *Core Animation Programming Guide*). You can scale the image and perform transformations, rotations and so on just as you would any layer. One difference is that you may need to set the layer's `orientation` property to specify how it should rotate images coming from the camera. In addition, on iPhone 4 the preview layer supports mirroring (this is the default when previewing the front-facing camera).

Video Gravity Modes

The preview layer supports three gravity modes that you set using `videoGravity`:

- `AVLayerVideoGravityResizeAspect`: This preserves the aspect ratio, leaving black bars where the video does not fill the available screen area.
- `AVLayerVideoGravityResizeAspectFill`: This preserves the aspect ratio, but fills the available screen area, cropping the video when necessary.
- `AVLayerVideoGravityResize`: This simply stretches the video to fill the available screen area, even if doing so distorts the image.

Using "Tap to Focus" With a Preview

You need to take care when implementing tap-to-focus in conjunction with a preview layer. You must account for the preview orientation and gravity of the layer, and the possibility that the preview may be mirrored.

Showing Audio Levels

To monitor the average and peak power levels in an audio channel in a capture connection, you use an `AVCaptureAudioChannel` object. Audio levels are not key-value observable, so you must poll for updated levels as often as you want to update your user interface (for example, 10 times a second).

```
AVCaptureAudioDataOutput *audioDataOutput = <#Get the audio data output#>;
NSArray *connections = audioDataOutput.connections;
if ([connections count] > 0) {
    // There should be only one connection to an AVCaptureAudioDataOutput.
    AVCaptureConnection *connection = [connections objectAtIndex:0];

    NSArray *audioChannels = connection.audioChannels;

    for (AVCaptureAudioChannel *channel in audioChannels) {
        float avg = channel.averagePowerLevel;
        float peak = channel.peakHoldLevel;
        // Update the level meter user interface.
    }
}
```

Putting it all Together: Capturing Video Frames as UIImage Objects

This brief code example to illustrates how you can capture video and convert the frames you get to `UIImage` objects. It shows you how to:

- Create an `AVCaptureSession` object to coordinate the flow of data from an AV input device to an output
- Find the `AVCaptureDevice` object for the input type you want
- Create an `AVCaptureDeviceInput` object for the device
- Create an `AVCaptureVideoDataOutput` object to produce video frames
- Implement a delegate for the `AVCaptureVideoDataOutput` object to process video frames
- Implement a function to convert the CMSampleBuffer received by the delegate into a `UIImage` object

> **Note** To focus on the most relevant code, this example omits several aspects of a complete application, including memory management. To use AV Foundation, you are expected to have enough experience with Cocoa to be able to infer the missing pieces.

Create and Configure a Capture Session

You use an `AVCaptureSession` object to coordinate the flow of data from an AV input device to an output. Create a session, and configure it to produce medium resolution video frames.

```
AVCaptureSession *session = [[AVCaptureSession alloc] init];
session.sessionPreset = AVCaptureSessionPresetMedium;
```

Create and Configure the Device and Device Input

Capture devices are represented by `AVCaptureDevice` objects; the class provides methods to retrieve an object for the input type you want. A device has one or more ports, configured using an `AVCaptureInput` object. Typically, you use the capture input in its default configuration.

Find a video capture device, then create a device input with the device and add it to the session.

```
AVCaptureDevice *device =
        [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeVideo];

NSError *error = nil;
AVCaptureDeviceInput *input =
```

```
        [AVCaptureDeviceInput deviceInputWithDevice:device error:&error];
if (!input) {
    // Handle the error appropriately.
}
[session addInput:input];
```

Create and Configure the Data Output

You use an `AVCaptureVideoDataOutput` object to process uncompressed frames from the video being captured. You typically configure several aspects of an output. For video, for example, you can specify the pixel format using the `videoSettings` property, and cap the frame rate by setting the `minFrameDuration` property.

Create and configure an output for video data and add it to the session; cap the frame rate to 15 fps by setting the `minFrameDuration` property to 1/15 second:

```
AVCaptureVideoDataOutput *output = [[[AVCaptureVideoDataOutput alloc] init]
autorelease];
[session addOutput:output];
output.videoSettings =
                [NSDictionary dictionaryWithObject:[NSNumber
numberWithInt:kCVPixelFormatType_32BGRA]

forKey:(id)kCVPixelBufferPixelFormatTypeKey];
output.minFrameDuration = CMTimeMake(1, 15);
```

The data output object uses delegation to vend the video frames. The delegate must adopt the `AVCaptureVideoDataOutputSampleBufferDelegate` protocol. When you set the data output's delegate, you must also provide a queue on which callbacks should be invoked.

```
dispatch_queue_t queue = dispatch_queue_create("MyQueue", NULL);
[output setSampleBufferDelegate:self queue:queue];
dispatch_release(queue);
```

You use the queue to modify the priority given to delivering and processing the video frames.

Implement the Sample Buffer Delegate Method

In the delegate class, implement the method (`captureOutput:didOutputSampleBuffer:fromConnection:`) that is called when a sample buffer is written. The video data output object delivers frames as CMSampleBuffers, so you need to convert from the CMSampleBuffer to a `UIImage` object. The function for this operation is shown in "Converting a CMSampleBuffer to a UIImage" (page 60).

```
- (void)captureOutput:(AVCaptureOutput *)captureOutput
         didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer
         fromConnection:(AVCaptureConnection *)connection {

    UIImage *image = imageFromSampleBuffer(sampleBuffer);
    // Add your code here that uses the image.
}
```

Remember that the delegate method is invoked on the queue you specified in `setSampleBufferDelegate:queue:`; if you want to update the user interface, you must invoke any relevant code on the main thread.

Starting and Stopping Recording

After configuring the capture session, you send it a `startRunning` message to start the recording.

```
[session startRunning];
```

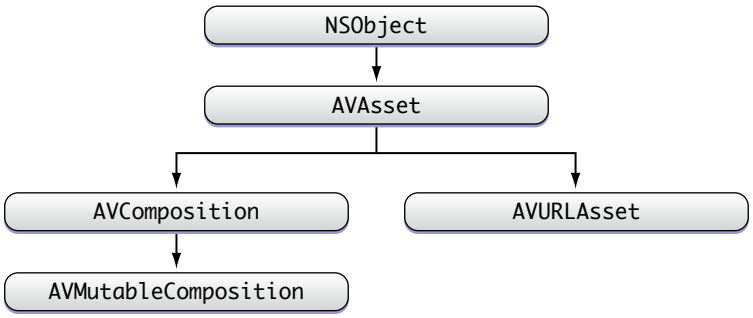
To stop recording, you send the session a `stopRunning` message.

Time and Media Representations

Time-based audio-visual data such as a movie file or a video stream is represented in the AV Foundation framework by `AVAsset`. Its structure dictates much of the framework works. Several low-level data structures that AV Foundation uses to represent time and media such as sample buffers come from the Core Media framework.

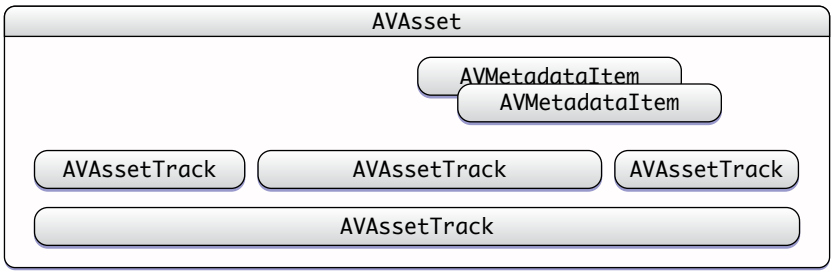
Representation of Assets

`AVAsset` is the core class in the AV Foundation framework. It provides a format-independent abstraction of time-based audiovisual data, such as a movie file or a video stream. In many cases, you work with one of its subclasses: you use the composition subclasses when you create new assets (see "Editing" (page 7)), and you use `AVURLAsset` to create a new asset instance from media at a given URL (including assets from the MPMedia framework or the Asset Library framework—see "Using Assets" (page 9)).



An asset contains a collection of tracks that are intended to be presented or processed together, each of a uniform media type, including (but not limited to) audio, video, text, closed captions, and subtitles. The asset object provides information about whole resource, such as its duration or title, as well as hints for presentation, such as its natural size. Assets may also have metadata, represented by instances of `AVMetadataItem`.

A track is represented by an instance of `AVAssetTrack`. In a typical simple case, one track represents the audio component and another represents the video component; in a complex composition, there may be multiple overlapping tracks of audio and video.



A track has a number of properties, such as its type (video or audio), visual and/or audible characteristics (as appropriate), metadata, and timeline (expressed in terms of its parent asset). A track also has an array of format descriptions. The array contains CMFormatDescriptions (see `CMFormatDescriptionRef`), each of which describes the format of media samples referenced by the track. A track that contains uniform media (for example, all encoded using to the same settings) will provide an array with a count of 1.

A track may itself be divided into segments, represented by instances of `AVAssetTrackSegment`. A segment is a time mapping from the source to the asset track timeline.

Representations of Time

Time in AV Foundation is represented by primitive structures from the Core Media framework.

CMTime Represents a Length of Time

`CMTime` is a C structure that represents time as a rational number, with a numerator (an `int64_t` value), and a denominator (an `int32_t` timescale).Conceptually, the timescale specifies the fraction of a second each unit in the numerator occupies. Thus if the timescale is 4, each unit represents a quarter of a second; if the timescale is 10, each unit represents a tenth of a second, and so on. You frequently use a timescale of 600, since this is a common multiple of several commonly-used frame-rates: 24 frames per second (fps) for film, 30 fps for NTSC (used for TV in North America and Japan), and 25 fps for PAL (used for TV in Europe). Using a timescale of 600, you can exactly represent any number of frames in these systems.

In addition to a simple time value, a CMTime can represent non-numeric values: +infinity, -infinity, and indefinite. It can also indicate whether the time been rounded at some point, and it maintains an epoch number.

Using CMTime

You create a time using `CMTimeMake`, or one of the related functions such as `CMTimeMakeWithSeconds` (which allows you to create a time using a float value and specify a preferred time scale). There are several functions for time-based arithmetic and to compare times, as illustrated in the following example.

```
CMTime time1 = CMTimeMake(200, 2); // 200 half-seconds
CMTime time2 = CMTimeMake(400, 4); // 400 quarter-seconds

// time1 and time2 both represent 100 seconds, but using different timescales.
if (CMTimeCompare(time1, time2) == 0) {
    NSLog(@"time1 and time2 are the same");
}

Float64 float64Seconds = 200.0 / 3;
CMTime time3 = CMTimeMakeWithSeconds(float64Seconds , 3); // 66.66... third-seconds
time3 = CMTimeMultiply(time3, 3);
// time3 now represents 200 seconds; next subtract time1 (100 seconds).
time3 = CMTimeSubtract(time3, time1);
CMTimeShow(time3);

if (CMTIME_COMPARE_INLINE(time2, ==, time3)) {
    NSLog(@"time2 and time3 are the same");
}
```

For a list of all the available functions, see *CMTime Reference*.

Special Values of CMTime

Core Media provides constants for special values: `kCMTimeZero`, `kCMTimeInvalid`, `kCMTimePositiveInfinity`, and `kCMTimeNegativeInfinity`. There are many ways, though in which a CMTime can, for example, represent a time that is invalid. If you need to test whether a CMTime is valid, or a non-numeric value, you should use an appropriate macro, such as `CMTIME_IS_INVALID`, `CMTIME_IS_POSITIVE_INFINITY`, or `CMTIME_IS_INDEFINITE`.

```
CMTime myTime = <#Get a CMTime#>;
if (CMTIME_IS_INVALID(myTime)) {
    // Perhaps treat this as an error; display a suitable alert to the user.
```

```
}
```

You should not compare the value of an arbitrary CMTime with `kCMTimeInvalid`.

Representing a CMTime as an Object

If you need to use CMTimes in annotations or Core Foundation containers, you can convert a CMTime to and from a CFDictionary (see `CFDictionaryRef`) using `CMTimeCopyAsDictionary` and `CMTimeMakeFromDictionary` respectively. You can also get a string representation of a CMTime using `CMTimeCopyDescription`.

Epochs

The epoch number of a CMTime is usually set to 0, but you can use it to distinguish unrelated timelines. For example, the epoch could be incremented each cycle through a presentation loop, to differentiate between time N in loop 0 from time N in loop 1.

CMTimeRange Represents a Time Range

`CMTimeRange` is a C structure that has a start time and duration, both expressed as CMTimes. A time range does not include the time that is the start time plus the duration.

You create a time range using `CMTimeRangeMake` or `CMTimeRangeFromTimeToTime`. There are constraints on the value of the CMTimes' epochs:

- CMTimeRanges cannot span different epochs.
- The epoch in a CMTime that represents a timestamp may be non-zero, but you can only perform range operations (such as `CMTimeRangeGetUnion`) on ranges whose start fields have the same epoch.
- The epoch in a CMTime that represents a duration should always be 0, and the value must be non-negative.

Working with Time Ranges

Core Media provides functions you can use to determine whether a time range contains a given time or other time range, or whether two time ranges are equal, and to calculate unions and intersections of time ranges, such as `CMTimeRangeContainsTime`, `CMTimeRangeEqual`, `CMTimeRangeContainsTimeRange`, and `CMTimeRangeGetUnion`.

Given that a time range does not include the time that is the start time plus the duration, the following expression always evaluates to false:

```
CMTimeRangeContainsTime(range, CMTimeRangeGetEnd(range))
```

For a list of all the available functions, see *CMTimeRange Reference*.

Special Values of CMTimeRange

Core Media provides constants for a zero-length range and an invalid range, `kCMTimeRangeZero` and `kCMTimeRangeInvalid` respectively. There are many ways, though in which a CMTimeRange can be invalid, or zero—or indefinite (if one of the CMTimes is indefinite. If you need to test whether a CMTimeRange is valid, zero, or indefinite, you should use an appropriate macro: `CMTIMERANGE_IS_VALID`, `CMTIMERANGE_IS_INVALID`, `CMTIMERANGE_IS_EMPTY`, or `CMTIMERANGE_IS_EMPTY`.

```
CMTimeRange myTimeRange = <#Get a CMTimeRange#>;
if (CMTIMERANGE_IS_EMPTY(myTimeRange)) {
    // The time range is zero.
}
```

You should not compare the value of an arbitrary CMTimeRange with `kCMTimeRangeInvalid`.

Representing a CMTimeRange as an Object

If you need to use CMTimeRanges in annotations or Core Foundation containers, you can convert a CMTimeRange to and from a CFDictionary (see `CFDictionaryRef`) using `CMTimeRangeCopyAsDictionary` and `CMTimeRangeMakeFromDictionary` respectively. You can also get a string representation of a CMTime using `CMTimeRangeCopyDescription`.

Representations of Media

Video data and its associated metadata is represented in AV Foundation by opaque objects from the Core Media framework. Core Media represents video data using CMSampleBuffer (see `CMSampleBufferRef`). CMSampleBuffer is a Core Foundation-style opaque type; an instance contains the sample buffer for a frame of video data as a Core Video pixel buffer (see `CVPixelBufferRef`). You access the pixel buffer from a sample buffer using `CMSampleBufferGetImageBuffer`:

```
CVPixelBufferRef pixelBuffer = CMSampleBufferGetImageBuffer(<#A CMSampleBuffer#>);
```

From the pixel buffer, you can access the actual video data. For an example, see "Converting a CMSampleBuffer to a UIImage" (page 60).

In addition to the video data, you can retrieve a number of other aspects of the video frame:

- **Timing information** You get accurate timestamps for both the original presentation time and the decode time using `CMSampleBufferGetPresentationTimeStamp` and `CMSampleBufferGetDecodeTimeStamp` respectively.
- **Format information** The format information is encapsulated in a CMFormatDescription object (see `CMFormatDescriptionRef`). From the format description, you can get for example the pixel type and video dimensions using `CMVideoFormatDescriptionGetCodecType` and `CMVideoFormatDescriptionGetDimensions` respectively.
- **Metadata** Metadata are stored in a dictionary as an **attachment**. You use `CMGetAttachment` to retrieve the dictionary:

```
CMSampleBufferRef sampleBuffer = <#Get a sample buffer#>;
CFDictionaryRef metadataDictionary =
    CMGetAttachment(sampleBuffer, CFSTR("MetadataDictionary", NULL);
if (metadataDictionary) {
    // Do something with the metadata.
}
```

Converting a CMSampleBuffer to a UIImage

The following function shows how you can convert a CMSampleBuffer to a `UIImage` object. You should consider your requirements carefully before using it. Performing the conversion is a comparatively expensive operation. It is appropriate to, for example, create a still image from a frame of video data taken every second or so. You should not use this as a means to manipulate every frame of video coming from a capture device in real time.

```
UIImage *imageFromSampleBuffer(CMSampleBufferRef sampleBuffer) {

    CVImageBufferRef imageBuffer = CMSampleBufferGetImageBuffer(sampleBuffer);
    // Lock the base address of the pixel buffer.
    CVPixelBufferLockBaseAddress(imageBuffer,0);

    // Get the number of bytes per row for the pixel buffer.
    size_t bytesPerRow = CVPixelBufferGetBytesPerRow(imageBuffer);
    // Get the pixel buffer width and height.
```

```
    size_t width = CVPixelBufferGetWidth(imageBuffer);
    size_t height = CVPixelBufferGetHeight(imageBuffer);

    // Create a device-dependent RGB color space.
    static CGColorSpaceRef colorSpace = NULL;
    if (colorSpace == NULL) {
        colorSpace = CGColorSpaceCreateDeviceRGB();
            if (colorSpace == NULL) {
            // Handle the error appropriately.
            return nil;
        }
    }

    // Get the base address of the pixel buffer.
    void *baseAddress = CVPixelBufferGetBaseAddress(imageBuffer);
    // Get the data size for contiguous planes of the pixel buffer.
    size_t bufferSize = CVPixelBufferGetDataSize(imageBuffer);

    // Create a Quartz direct-access data provider that uses data we supply.
    CGDataProviderRef dataProvider =
        CGDataProviderCreateWithData(NULL, baseAddress, bufferSize, NULL);
    // Create a bitmap image from data supplied by the data provider.
    CGImageRef cgImage =
        CGImageCreate(width, height, 8, 32, bytesPerRow,
                      colorSpace, kCGImageAlphaNoneSkipFirst |
kCGBitmapByteOrder32Little,
                      dataProvider, NULL, true, kCGRenderingIntentDefault);
    CGDataProviderRelease(dataProvider);

    // Create and return an image object to represent the Quartz image.
    UIImage *image = [UIImage imageWithCGImage:cgImage];
    CGImageRelease(cgImage);

    CVPixelBufferUnlockBaseAddress(imageBuffer, 0);
```

```
    return image;
}
```

Document Revision History

This table describes the changes to *AV Foundation Programming Guide*.

Date	Notes
2011-10-12	Updated for iOS5 to include references to release notes.
2011-04-28	First release for Mac OS X v10.7.
2010-11-15	Added article describing capture.
2010-09-08	TBD
2010-08-16	First version of a document that describes a low-level framework you use to play, inspect, create, edit, capture, and transcode media assets.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010