AV Foundation Programming Guide

Developer

Contents

Figures and Listings

About AV Foundation

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.

Figure I-1 AV Foundation stack on iOS

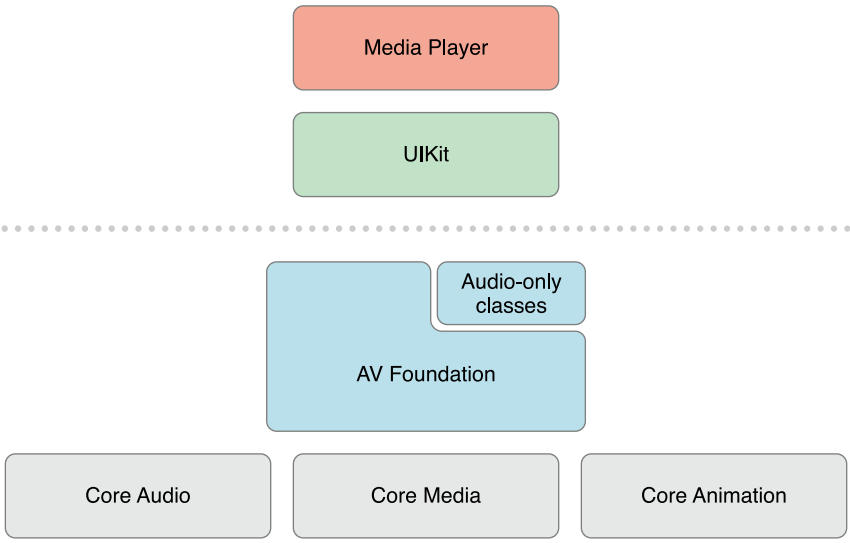
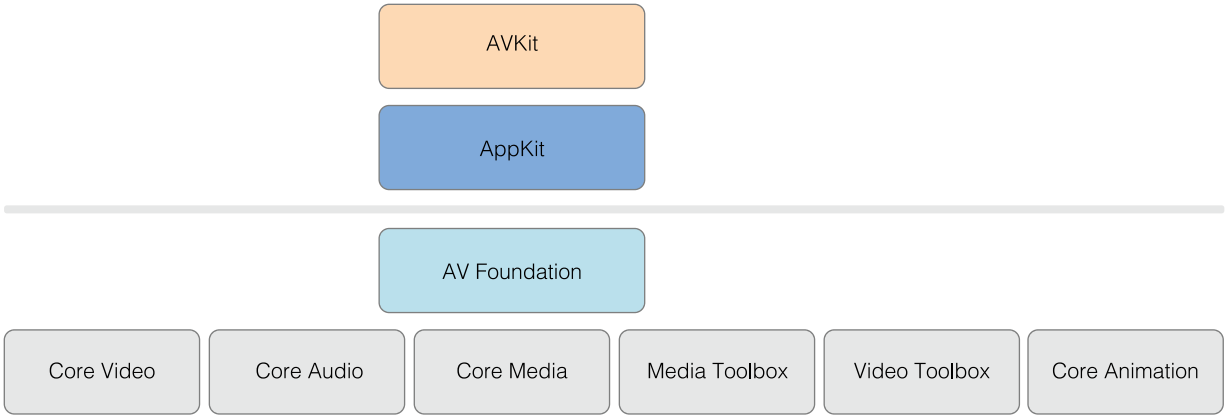


Figure I-2 AVFoundation stack on OS X



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, on iOS you use the Media Player framework (`MPMoviePlayerController` or `MPMoviePlayerViewController`), or for web-based media you could use a `UIWebView` object. On OS X use AV Kit Framework's `AVPlayerView` class to play a video asset.
- 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 4.0 and later, and OS X 10.7 and later.

At a Glance

There are two facets to the AV Foundation framework—APIs related to video and APIs related just to audio. The older audio-related classes provide easy ways to deal with audio. They are described in the *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*.

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 12)) and to create new compositions (see Editing (page 9)).

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 12), Time and Media Representations (page 104)

Playback

AV Foundation 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 track within an asset is managed by a **player item track** object. 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 the Core Animation layer. You can use a **player queue** to schedule playback of a collection of player items in sequence.

Relevant Chapter: Playback (page 21)

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—in 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, in 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 Chapter: Using Assets (page 12)

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 tracks to generate images.

Relevant Chapter: Using Assets (page 12)

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.

You can also create an asset from media such as sample buffers or still images using an **asset writer**.

Relevant Chapter: Editing (page 35)

Still and Video Media Capture

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 a **preview layer** to show the user what a camera is recording.

Relevant Chapter: Media Capture (page 50)

Concurrent Programming with AV Foundation

Callbacks from AV Foundation—invocations of blocks, key-value observers, and notification handlers—are not generally 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.

There are two general guidelines as far as notifications and threading:

- UI related notifications occur on the main thread.
- Classes or methods that require you create and/or specify a queue will return notifications on that queue.

Beyond those two guidelines (and there are exceptions, which are noted in the reference documentation) you should not assume that a notification will be returned on any specific thread.

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` 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*. The *AVCam for iOS* sample code is considered the primary example for all AV Foundation functionality and can be consulted for examples of thread and queue usage with AV Foundation.

Prerequisites

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* or, for basic playback, the *AV Kit Framework Reference*.

See Also

There are several AV Foundation examples including two that are key to understanding and implementation Camera capture functionality:

AVCam for iOS is the canonical sample code for implementing any program that uses the camera functionality. It is a complete sample, well documented, and covers the majority of the functionality showing the best practices.

AVCamManual: Using the Manual Capture API is the companion application to AVCam. It implements Camera functionality using the manual camera controls. It is also a complete example, well documented, and should be considered the canonical example for creating camera applications that take advantage of manual controls.

RosyWriter is an example that demonstrates real time frame processing and in particular how to apply filters to video content. This is a very common developer requirement and this example covers that functionality.

AVLocationPlayer: Using AVFoundation Metadata Reading APIs demonstrates using the metadata APIs.

Using Assets

Assets can come from a file or from media in the user's iPod library or Photo library. When you create an asset object all the information that you might want to retrieve for that item is not 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

The `AVURLAsset` 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 = @{ AVURLAssetPreferPreciseDurationAndTimingKey :
 @YES };
AVURLAsset *anAssetToUseInAComposition = [[AVURLAsset alloc]
initWithURL:url options:options];
```

Accessing the User's Assets

To access the assets managed by 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");
                }];
```

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 = @[@"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 21).

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 tracksWithMediaType:AVMediaTypeVideo] count] > 0) {
    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 keep a strong reference to 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;

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);


    // Do something interesting with the image.

    CGImageRelease(halfWayImage);

}
```

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` structure, 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, check the result constant to determine whether the image was created. In addition, ensure that you keep a strong reference to the image generator until it has finished creating the images.

```
AVAsset *myAsset = <#An asset#>];
// Assume: @property (strong) 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 = @[NSValue valueWithCMTime:kCMTimeZero],
                  [NSValue valueWithCMTime:firstThird], [NSValue
valueWithCMTime:secondThird],
                  [NSValue valueWithCMTime:end]];

[imageGenerator generateCGImagesAsynchronouslyForTimes:times
               completionHandler:^(CMTime requestedTime, CGImageRef image, CMTime
 actualTime,
                                   AVAssetImageGeneratorResult result, NSError
*error) {

                NSString *requestedTimeString = (NSString *)
                   CFBridgingRelease(CMTimeCopyDescription(NULL, requestedTime));
                NSString *actualTimeString = (NSString *)
```

```
                    CFBridgingRelease(CMTimeCopyDescription(NULL, actualTime));
                NSLog(@"Requested: %@; actual %@", requestedTimeString,
actualTimeString);

                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 the 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` value 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;
        }
    }];
```

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.

Playback

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

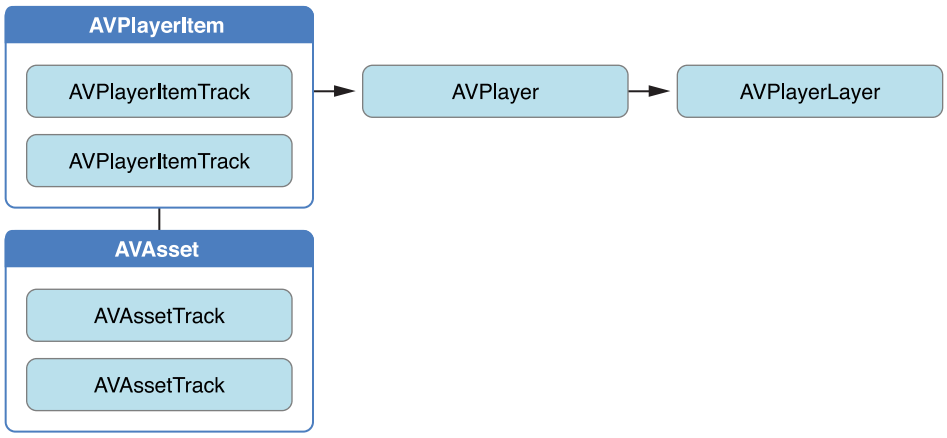
Playing Assets

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. You can use an `AVQueuePlayer` object to play a number of items in sequence (`AVQueuePlayer` is a subclass of `AVPlayer`). On OS X you have the option of the using the AV Kit framework's `AVPlayerView` class to play the content back within a view.

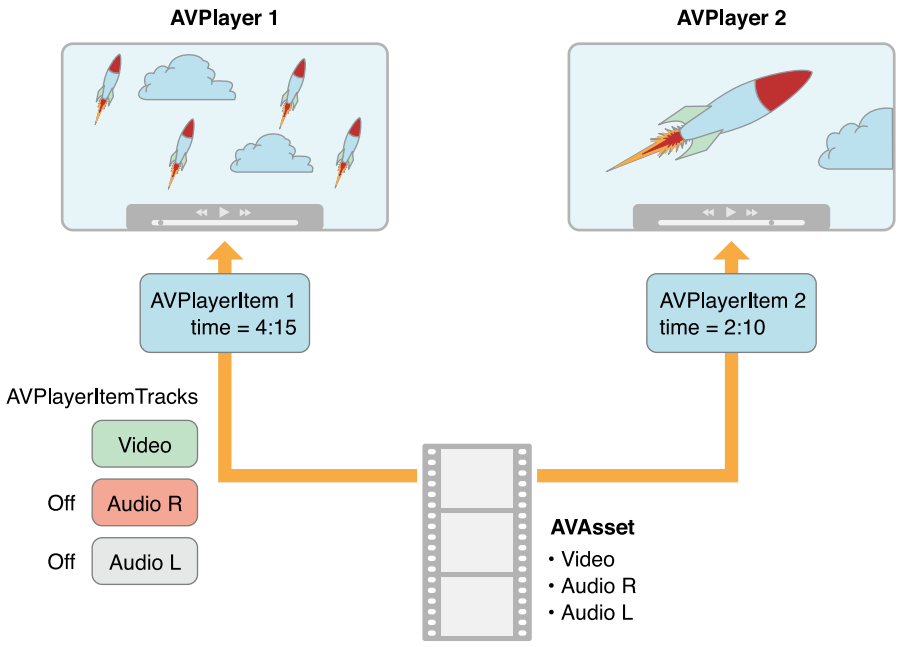
A player provides you with information about the state of the playback so, if you need to, you can synchronize your user interface with the player's state. You typically direct the output of a player to a specialized Core Animation layer (an instance of `AVPlayerLayer` or `AVSynchronizedLayer`). To learn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

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

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 (for example, 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 assets (HTTP Live Streaming format).

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

- Create an asset using `AVURLAsset`.
- Create an instance of `AVPlayerItem` using the asset.
- Associate the item with an instance of `AVPlayer`.
- Wait until the item's `status` property 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 29).

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.

To get the duration of a streaming item, you can observe the `duration` property on the player item. When the item becomes ready to play, this property updates to the correct value for the stream.

> **Note:** Using the `duration` property on the player item requires iOS 4.3 or later. An approach that is compatible with all versions of iOS involves observing the `status` property of the player item. When the status becomes `AVPlayerItemStatusReadyToPlay`, the duration can be fetched with the following line of code:
>
> ```
> [[[[playerItem tracks] objectAtIndex:0] assetTrack] asset] duration];
> ```

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

```
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 27).

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`.

Items that support reverse playback can use the rate property with a negative number to set the reverse playback rate. You determine the type of reverse play that is supported by using the playerItem properties `canPlayReverse` (supports a rate value of -1.0), `canPlaySlowReverse` (supports rates between 0.0 and 1.0) and `canPlayFastReverse` (supports rate values less than -1.0).

Seeking—Repositioning the Playhead

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

```
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:` as in the following code fragment:

```
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 use zero only 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 playhead back at the beginning of the item, you can register to receive an `AVPlayerItemDidPlayToEndTimeNotification` notification 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

You can use an `AVQueuePlayer` object to play a number of items in sequence. The `AVQueuePlayer` class 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 both 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` property 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` 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 then 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, if time jumps, and when playback starts or stops.
- With `addBoundaryTimeObserverForTimes:queue:usingBlock:`, you pass an array of `CMTime` structures 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 keep a strong reference to 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 (strong) 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 = @[[NSValue valueWithCMTime:firstThird], [NSValue
valueWithCMTime:secondThird]];


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


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

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 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 and 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 21).

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) 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) AVPlayer *player;
@property (nonatomic) AVPlayerItem *playerItem;
@property (nonatomic, weak) IBOutlet PlayerView *playerView;
@property (nonatomic, weak) 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 ((self.player.currentItem != nil) &&
        ([self.player.currentItem status] == AVPlayerItemStatusReadyToPlay)) {
        self.playButton.enabled = YES;
    }
    else {
        self.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`. (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:@[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` property. You should configure this observing before associating the player item instance with the player itself.

You trigger the player item's 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;
                AVKeyValueStatus status = [asset statusOfValueForKey:tracksKey
error:&error];

                if (status == AVKeyValueStatusLoaded) {
```

```
                    self.playerItem = [AVPlayerItem playerItemWithAsset:asset];
                   // ensure that this is done before the playerItem is associated
 with the player
                    [self.playerItem addObserver:self forKeyPath:@"status"
                                         options:NSKeyValueObservingOptionInitial
context:&ItemStatusContext];
                    [[NSNotificationCenter defaultCenter] addObserver:self

selector:@selector(playerItemDidReachEnd:)

name:AVPlayerItemDidPlayToEndTimeNotification

object:self.playerItem];
                   self.player = [AVPlayer playerWithPlayerItem:self.playerItem];
                    [self.playerView setPlayer:self.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` 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 involves sending a `play` message to the player.

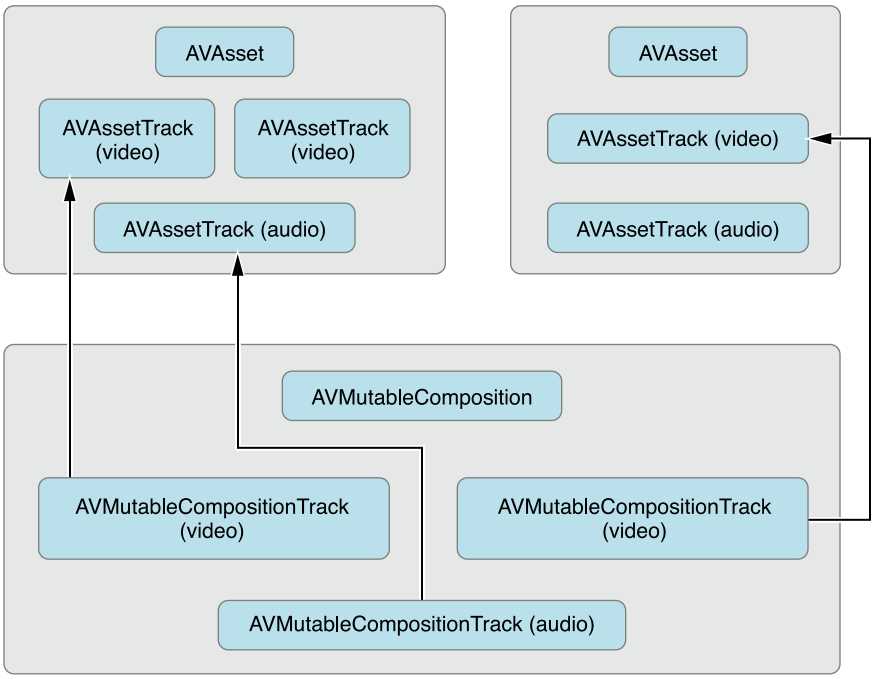
```
- (IBAction)play:sender {
    [player play];
}
```

The item is played only once. After playback, the player's head is set to the end of the item, and further invocations of the `play` method will have no effect. To position the playhead 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`.

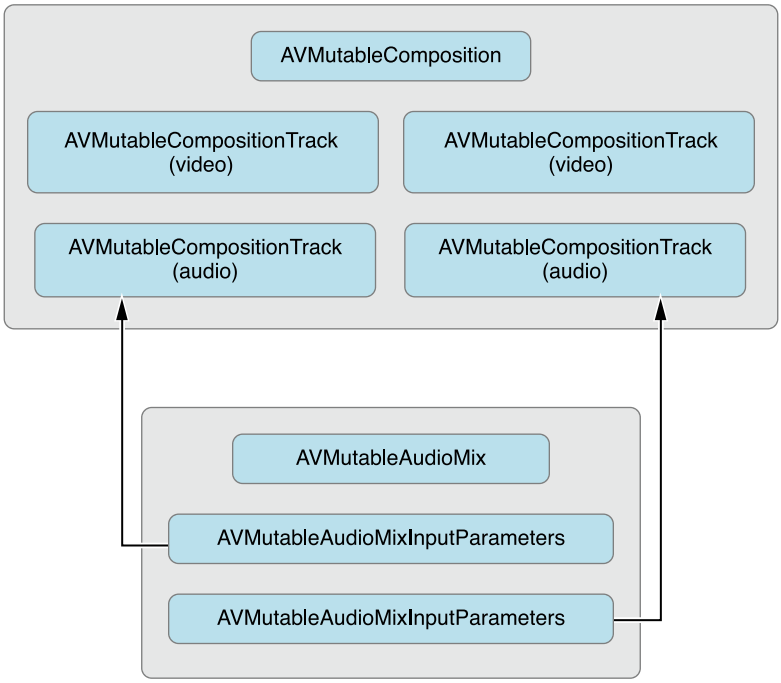
```
// Register with the notification center after creating the player item.
    [[NSNotificationCenter defaultCenter]
        addObserver:self
        selector:@selector(playerItemDidReachEnd:)
        name:AVPlayerItemDidPlayToEndTimeNotification
        object:[self.player currentItem]];

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

Editing

The AV Foundation framework provides a feature-rich set of classes to facilitate the editing of audio visual assets. At the heart of AV Foundation's editing API are compositions. A composition is simply a collection of tracks from one or more different media assets. The `AVMutableComposition` class provides an interface for inserting and removing tracks, as well as managing their temporal orderings. You use a mutable composition to piece together a new asset from a combination of existing assets. If all you want to do is merge multiple assets together sequentially into a single file, that is as much detail as you need. If you want to perform any custom audio or video processing on the tracks in your composition, you need to incorporate an audio mix or a video composition, respectively.

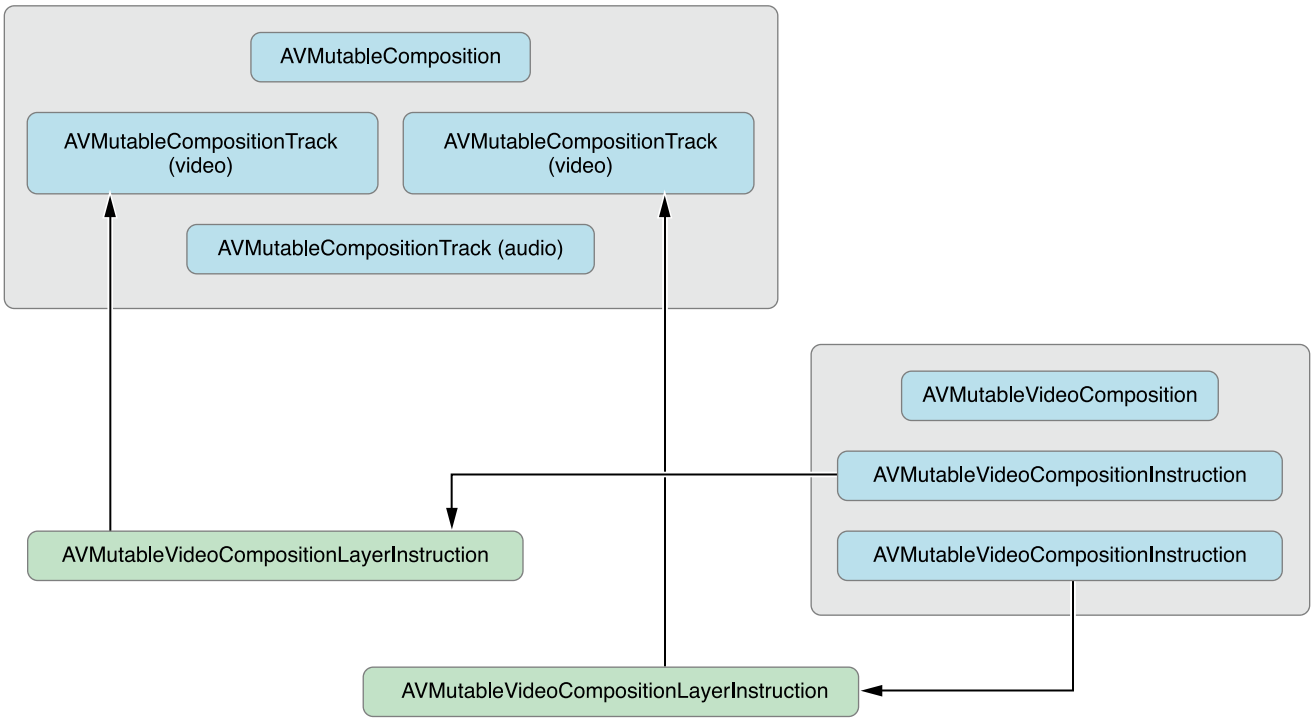


Using the `AVMutableAudioMix` class, you can perform custom audio processing on the audio tracks in your composition. Currently, you can specify a maximum volume or set a volume ramp for an audio track.

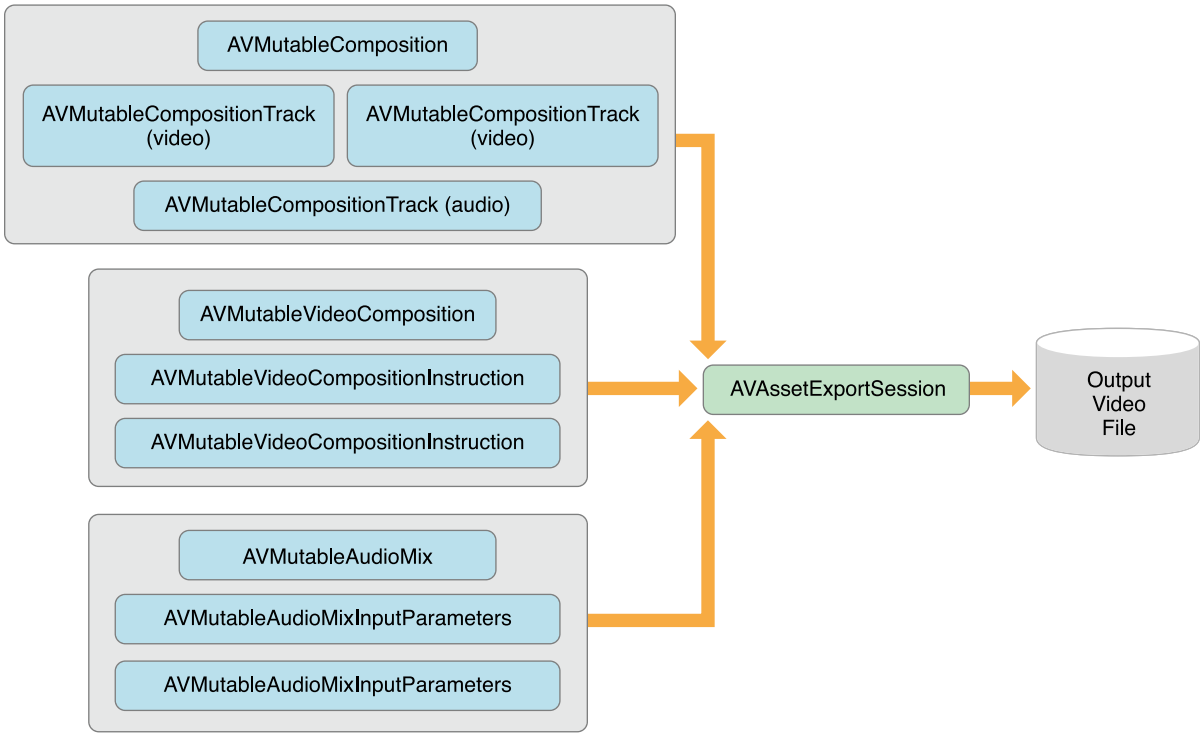


You can use the `AVMutableVideoComposition` class to work directly with the video tracks in your composition for the purposes of editing. With a single video composition, you can specify the desired render size and scale, as well as the frame duration, for the output video. Through a video composition's instructions (represented by the `AVMutableVideoCompositionInstruction` class), you can modify the background color of your video and apply layer instructions. These layer instructions (represented by the `AVMutableVideoCompositionLayerInstruction` class) can be used to apply transforms, transform ramps,

opacity and opacity ramps to the video tracks within your composition. The video composition class also gives you the ability to introduce effects from the Core Animation framework into your video using the `animationTool` property.



To combine your composition with an audio mix and a video composition, you use an `AVAssetExportSession` object. You initialize the export session with your composition and then simply assign your audio mix and video composition to the `audioMix` and `videoComposition` properties respectively.



Creating a Composition

To create your own composition, you use the `AVMutableComposition` class. To add media data to your composition, you must add one or more composition tracks, represented by the `AVMutableCompositionTrack` class. The simplest case is creating a mutable composition with one video track and one audio track:

```
AVMutableComposition *mutableComposition = [AVMutableComposition composition];

// Create the video composition track.

AVMutableCompositionTrack *mutableCompositionVideoTrack = [mutableComposition
addMutableTrackWithMediaType:AVMediaTypeVideo
preferredTrackID:kCMPersistentTrackID_Invalid];

// Create the audio composition track.

AVMutableCompositionTrack *mutableCompositionAudioTrack = [mutableComposition
addMutableTrackWithMediaType:AVMediaTypeAudio
preferredTrackID:kCMPersistentTrackID_Invalid];
```

Options for Initializing a Composition Track

When adding new tracks to a composition, you must provide both a media type and a track ID. Although audio and video are the most commonly used media types, you can specify other media types as well, such as `AVMediaTypeSubtitle` or `AVMediaTypeText`.

Every track associated with some audiovisual data has a unique identifier referred to as a track ID. If you specify `kCMPersistentTrackID_Invalid` as the preferred track ID, a unique identifier is automatically generated for you and associated with the track.

Adding Audiovisual Data to a Composition

Once you have a composition with one or more tracks, you can begin adding your media data to the appropriate tracks. To add media data to a composition track, you need access to the `AVAsset` object where the media data is located. You can use the mutable composition track interface to place multiple tracks with the same underlying media type together on the same track. The following example illustrates how to add two different video asset tracks in sequence to the same composition track:

```
// You can retrieve AVAssets from a number of places, like the camera roll for
example.

AVAsset *videoAsset = <#AVAsset with at least one video track#>;

AVAsset *anotherVideoAsset = <#another AVAsset with at least one video track#>;

// Get the first video track from each asset.

AVAssetTrack *videoAssetTrack = [[videoAsset tracksWithMediaType:AVMediaTypeVideo]
 objectAtIndex:0];

AVAssetTrack *anotherVideoAssetTrack = [[anotherVideoAsset
tracksWithMediaType:AVMediaTypeVideo] objectAtIndex:0];

// Add them both to the composition.

[mutableCompositionVideoTrack
insertTimeRange:CMTimeRangeMake(kCMTimeZero,videoAssetTrack.timeRange.duration)
ofTrack:videoAssetTrack atTime:kCMTimeZero error:nil];

[mutableCompositionVideoTrack
insertTimeRange:CMTimeRangeMake(kCMTimeZero,anotherVideoAssetTrack.timeRange.duration)
 ofTrack:anotherVideoAssetTrack atTime:videoAssetTrack.timeRange.duration error:nil];
```

Retrieving Compatible Composition Tracks

Where possible, you should have only one composition track for each media type. This unification of compatible asset tracks leads to a minimal amount of resource usage. When presenting media data serially, you should place any media data of the same type on the same composition track. You can query a mutable composition to find out if there are any composition tracks compatible with your desired asset track:

```
AVMutableCompositionTrack *compatibleCompositionTrack = [mutableComposition
mutableTrackCompatibleWithTrack:<#the AVAssetTrack you want to insert#>];
if (compatibleCompositionTrack) {
    // Implementation continues.
}
```

> **Note:** Placing multiple video segments on the same composition track can potentially lead to dropping frames at the transitions between video segments, especially on embedded devices. Choosing the number of composition tracks for your video segments depends entirely on the design of your app and its intended platform.

Generating a Volume Ramp

A single `AVMutableAudioMix` object can perform custom audio processing on all of the audio tracks in your composition individually. You create an audio mix using the `audioMix` class method, and you use instances of the `AVMutableAudioMixInputParameters` class to associate the audio mix with specific tracks within your composition. An audio mix can be used to vary the volume of an audio track. The following example displays how to set a volume ramp on a specific audio track to slowly fade the audio out over the duration of the composition:

```
AVMutableAudioMix *mutableAudioMix = [AVMutableAudioMix audioMix];
// Create the audio mix input parameters object.
AVMutableAudioMixInputParameters *mixParameters = [AVMutableAudioMixInputParameters
 audioMixInputParametersWithTrack:mutableCompositionAudioTrack];
// Set the volume ramp to slowly fade the audio out over the duration of the
composition.
[mixParameters setVolumeRampFromStartVolume:1.f toEndVolume:0.f
timeRange:CMTimeRangeMake(kCMTimeZero, mutableComposition.duration)];
// Attach the input parameters to the audio mix.
mutableAudioMix.inputParameters = @[mixParameters];
```

Performing Custom Video Processing

As with an audio mix, you only need one `AVMutableVideoComposition` object to perform all of your custom video processing on your composition's video tracks. Using a video composition, you can directly set the appropriate render size, scale, and frame rate for your composition's video tracks. For a detailed example of setting appropriate values for these properties, see Setting the Render Size and Frame Duration (page 47).

Changing the Composition's Background Color

All video compositions must also have an array of `AVVideoCompositionInstruction` objects containing at least one video composition instruction. You use the `AVMutableVideoCompositionInstruction` class to create your own video composition instructions. Using video composition instructions, you can modify the composition's background color, specify whether post processing is needed or apply layer instructions.

The following example illustrates how to create a video composition instruction that changes the background color to red for the entire composition.

```
AVMutableVideoCompositionInstruction *mutableVideoCompositionInstruction =
[AVMutableVideoCompositionInstruction videoCompositionInstruction];

mutableVideoCompositionInstruction.timeRange = CMTimeRangeMake(kCMTimeZero,
mutableComposition.duration);

mutableVideoCompositionInstruction.backgroundColor = [[UIColor redColor] CGColor];
```

Applying Opacity Ramps

Video composition instructions can also be used to apply video composition layer instructions. An `AVMutableVideoCompositionLayerInstruction` object can apply transforms, transform ramps, opacity and opacity ramps to a certain video track within a composition. The order of the layer instructions in a video composition instruction's `layerInstructions` array determines how video frames from source tracks should be layered and composed for the duration of that composition instruction. The following code fragment shows how to set an opacity ramp to slowly fade out the first video in a composition before transitioning to the second video:

```
AVAsset *firstVideoAssetTrack = <#AVAssetTrack representing the first video segment
 played in the composition#>;

AVAsset *secondVideoAssetTrack = <#AVAssetTrack representing the second video
segment played in the composition#>;

// Create the first video composition instruction.

AVMutableVideoCompositionInstruction *firstVideoCompositionInstruction =
[AVMutableVideoCompositionInstruction videoCompositionInstruction];

// Set its time range to span the duration of the first video track.
```

```
firstVideoCompositionInstruction.timeRange = CMTimeRangeMake(kCMTimeZero,
firstVideoAssetTrack.timeRange.duration);

// Create the layer instruction and associate it with the composition video track.

AVMutableVideoCompositionLayerInstruction *firstVideoLayerInstruction =
[AVMutableVideoCompositionLayerInstruction
videoCompositionLayerInstructionWithAssetTrack:mutableCompositionVideoTrack];

// Create the opacity ramp to fade out the first video track over its entire
duration.

[firstVideoLayerInstruction setOpacityRampFromStartOpacity:1.f toEndOpacity:0.f
timeRange:CMTimeRangeMake(kCMTimeZero, firstVideoAssetTrack.timeRange.duration)];

// Create the second video composition instruction so that the second video track
 isn't transparent.

AVMutableVideoCompositionInstruction *secondVideoCompositionInstruction =
[AVMutableVideoCompositionInstruction videoCompositionInstruction];

// Set its time range to span the duration of the second video track.

secondVideoCompositionInstruction.timeRange =
CMTimeRangeMake(firstVideoAssetTrack.timeRange.duration,
CMTimeAdd(firstVideoAssetTrack.timeRange.duration,
secondVideoAssetTrack.timeRange.duration));

// Create the second layer instruction and associate it with the composition video
 track.

AVMutableVideoCompositionLayerInstruction *secondVideoLayerInstruction =
[AVMutableVideoCompositionLayerInstruction
videoCompositionLayerInstructionWithAssetTrack:mutableCompositionVideoTrack];

// Attach the first layer instruction to the first video composition instruction.

firstVideoCompositionInstruction.layerInstructions = @[firstVideoLayerInstruction];

// Attach the second layer instruction to the second video composition instruction.

secondVideoCompositionInstruction.layerInstructions = @[secondVideoLayerInstruction];

// Attach both of the video composition instructions to the video composition.

mutableVideoComposition.instructions = @[firstVideoCompositionInstruction,
secondVideoCompositionInstruction];
```

Incorporating Core Animation Effects

A video composition can add the power of Core Animation to your composition through the `animationTool` property. Through this animation tool, you can accomplish tasks such as watermarking video and adding titles or animating overlays. Core Animation can be used in two different ways with video compositions: You can add a Core Animation layer in as its own individual composition track, or you can render Core Animation effects (using a Core Animation layer) into the video frames in your composition directly. The following code displays the latter option by adding a watermark to the center of the video:

```
CALayer *watermarkLayer = <#CALayer representing your desired watermark image#>;

CALayer *parentLayer = [CALayer layer];

CALayer *videoLayer = [CALayer layer];

parentLayer.frame = CGRectMake(0, 0, mutableVideoComposition.renderSize.width,
mutableVideoComposition.renderSize.height);

videoLayer.frame = CGRectMake(0, 0, mutableVideoComposition.renderSize.width,
mutableVideoComposition.renderSize.height);

[parentLayer addSublayer:videoLayer];

watermarkLayer.position = CGPointMake(mutableVideoComposition.renderSize.width/2,
 mutableVideoComposition.renderSize.height/4);

[parentLayer addSublayer:watermarkLayer];

mutableVideoComposition.animationTool = [AVVideoCompositionCoreAnimationTool
videoCompositionCoreAnimationToolWithPostProcessingAsVideoLayer:videoLayer
inLayer:parentLayer];
```

Putting It All Together: Combining Multiple Assets and Saving the Result to the Camera Roll

This brief code example illustrates how you can combine two video asset tracks and an audio asset track to create a single video file. It shows how to:

- Create an `AVMutableComposition` object and add multiple `AVMutableCompositionTrack` objects
- Add time ranges of `AVAssetTrack` objects to compatible composition tracks
- Check the `preferredTransform` property of a video asset track to determine the video's orientation
- Use `AVMutableVideoCompositionLayerInstruction` objects to apply transforms to the video tracks within a composition
- Set appropriate values for the `renderSize` and `frameDuration` properties of a video composition
- Use a composition in conjunction with a video composition when exporting to a video file
- Save a video file to the Camera Roll

Note: To focus on the most relevant code, this example omits several aspects of a complete app, such as memory management and error handling. To use AV Foundation, you are expected to have enough experience with Cocoa to infer the missing pieces.

Creating the Composition

To piece together tracks from separate assets, you use an `AVMutableComposition` object. Create the composition and add one audio and one video track.

```
AVMutableComposition *mutableComposition = [AVMutableComposition composition];

AVMutableCompositionTrack *videoCompositionTrack = [mutableComposition
addMutableTrackWithMediaType:AVMediaTypeVideo
preferredTrackID:kCMPersistentTrackID_Invalid];

AVMutableCompositionTrack *audioCompositionTrack = [mutableComposition
addMutableTrackWithMediaType:AVMediaTypeAudio
preferredTrackID:kCMPersistentTrackID_Invalid];
```

Adding the Assets

An empty composition does you no good. Add the two video asset tracks and the audio asset track to the composition.

```
AVAssetTrack *firstVideoAssetTrack = [[firstVideoAsset
tracksWithMediaType:AVMediaTypeVideo] objectAtIndex:0];

AVAssetTrack *secondVideoAssetTrack = [[secondVideoAsset
tracksWithMediaType:AVMediaTypeVideo] objectAtIndex:0];

[videoCompositionTrack insertTimeRange:CMTimeRangeMake(kCMTimeZero,
firstVideoAssetTrack.timeRange.duration) ofTrack:firstVideoAssetTrack
atTime:kCMTimeZero error:nil];

[videoCompositionTrack insertTimeRange:CMTimeRangeMake(kCMTimeZero,
secondVideoAssetTrack.timeRange.duration) ofTrack:secondVideoAssetTrack
atTime:firstVideoAssetTrack.timeRange.duration error:nil];

[audioCompositionTrack insertTimeRange:CMTimeRangeMake(kCMTimeZero,
CMTimeAdd(firstVideoAssetTrack.timeRange.duration,
secondVideoAssetTrack.timeRange.duration)) ofTrack:[[audioAsset
tracksWithMediaType:AVMediaTypeAudio] objectAtIndex:0] atTime:kCMTimeZero error:nil];
```

Note: This assumes that you have two assets that contain at least one video track each and a third asset that contains at least one audio track. The videos can be retrieved from the Camera Roll, and the audio track can be retrieved from the music library or the videos themselves.

Checking the Video Orientations

Once you add your video and audio tracks to the composition, you need to ensure that the orientations of both video tracks are correct. By default, all video tracks are assumed to be in landscape mode. If your video track was taken in portrait mode, the video will not be oriented properly when it is exported. Likewise, if you try to combine a video shot in portrait mode with a video shot in landscape mode, the export session will fail to complete.

```
BOOL isFirstVideoPortrait = NO;

CGAffineTransform firstTransform = firstVideoAssetTrack.preferredTransform;

// Check the first video track's preferred transform to determine if it was recorded
 in portrait mode.

if (firstTransform.a == 0 && firstTransform.d == 0 && (firstTransform.b == 1.0 ||
 firstTransform.b == -1.0) && (firstTransform.c == 1.0 || firstTransform.c ==
-1.0)) {

    isFirstVideoPortrait = YES;

}

BOOL isSecondVideoPortrait = NO;

CGAffineTransform secondTransform = secondVideoAssetTrack.preferredTransform;

// Check the second video track's preferred transform to determine if it was
recorded in portrait mode.

if (secondTransform.a == 0 && secondTransform.d == 0 && (secondTransform.b == 1.0
 || secondTransform.b == -1.0) && (secondTransform.c == 1.0 || secondTransform.c
== -1.0)) {

    isSecondVideoPortrait = YES;

}

if ((isFirstVideoAssetPortrait && !isSecondVideoAssetPortrait) ||
(!isFirstVideoAssetPortrait && isSecondVideoAssetPortrait)) {

    UIAlertView *incompatibleVideoOrientationAlert = [[UIAlertView alloc]
initWithTitle:@"Error!" message:@"Cannot combine a video shot in portrait mode
with a video shot in landscape mode." delegate:self cancelButtonTitle:@"Dismiss"
otherButtonTitles:nil];

    [incompatibleVideoOrientationAlert show];

    return;

}
```

Applying the Video Composition Layer Instructions

Once you know the video segments have compatible orientations, you can apply the necessary layer instructions to each one and add these layer instructions to the video composition.

```
AVMutableVideoCompositionInstruction *firstVideoCompositionInstruction =
[AVMutableVideoCompositionInstruction videoCompositionInstruction];

// Set the time range of the first instruction to span the duration of the first
video track.

firstVideoCompositionInstruction.timeRange = CMTimeRangeMake(kCMTimeZero,
firstVideoAssetTrack.timeRange.duration);

AVMutableVideoCompositionInstruction * secondVideoCompositionInstruction =
[AVMutableVideoCompositionInstruction videoCompositionInstruction];

// Set the time range of the second instruction to span the duration of the second
 video track.

secondVideoCompositionInstruction.timeRange =
CMTimeRangeMake(firstVideoAssetTrack.timeRange.duration,
CMTimeAdd(firstVideoAssetTrack.timeRange.duration,
secondVideoAssetTrack.timeRange.duration));

AVMutableVideoCompositionLayerInstruction *firstVideoLayerInstruction =
[AVMutableVideoCompositionLayerInstruction
videoCompositionLayerInstructionWithAssetTrack:videoCompositionTrack];

// Set the transform of the first layer instruction to the preferred transform of
 the first video track.

[firstVideoLayerInstruction setTransform:firstTransform atTime:kCMTimeZero];

AVMutableVideoCompositionLayerInstruction *secondVideoLayerInstruction =
[AVMutableVideoCompositionLayerInstruction
videoCompositionLayerInstructionWithAssetTrack:videoCompositionTrack];

// Set the transform of the second layer instruction to the preferred transform
of the second video track.

[secondVideoLayerInstruction setTransform:secondTransform
atTime:firstVideoAssetTrack.timeRange.duration];

firstVideoCompositionInstruction.layerInstructions = @[firstVideoLayerInstruction];

secondVideoCompositionInstruction.layerInstructions = @[secondVideoLayerInstruction];

AVMutableVideoComposition *mutableVideoComposition = [AVMutableVideoComposition
videoComposition];

mutableVideoComposition.instructions = @[firstVideoCompositionInstruction,
secondVideoCompositionInstruction];
```

All `AVAssetTrack` objects have a `preferredTransform` property that contains the orientation information for that asset track. This transform is applied whenever the asset track is displayed onscreen. In the previous code, the layer instruction's transform is set to the asset track's transform so that the video in the new composition displays properly once you adjust its render size.

Setting the Render Size and Frame Duration

To complete the video orientation fix, you must adjust the `renderSize` property accordingly. You should also pick a suitable value for the `frameDuration` property, such as 1/30th of a second (or 30 frames per second). By default, the `renderScale` property is set to `1.0`, which is appropriate for this composition.

```
CGSize naturalSizeFirst, naturalSizeSecond;

// If the first video asset was shot in portrait mode, then so was the second one
 if we made it here.

if (isFirstVideoAssetPortrait) {

// Invert the width and height for the video tracks to ensure that they display
properly.

    naturalSizeFirst = CGSizeMake(firstVideoAssetTrack.naturalSize.height,
firstVideoAssetTrack.naturalSize.width);

    naturalSizeSecond = CGSizeMake(secondVideoAssetTrack.naturalSize.height,
secondVideoAssetTrack.naturalSize.width);

}

else {

// If the videos weren't shot in portrait mode, we can just use their natural
sizes.

    naturalSizeFirst = firstVideoAssetTrack.naturalSize;

    naturalSizeSecond = secondVideoAssetTrack.naturalSize;

}

float renderWidth, renderHeight;

// Set the renderWidth and renderHeight to the max of the two videos widths and
heights.

if (naturalSizeFirst.width > naturalSizeSecond.width) {

    renderWidth = naturalSizeFirst.width;

}

else {

    renderWidth = naturalSizeSecond.width;

}

if (naturalSizeFirst.height > naturalSizeSecond.height) {

    renderHeight = naturalSizeFirst.height;
```

```
}
else {
    renderHeight = naturalSizeSecond.height;
}
mutableVideoComposition.renderSize = CGSizeMake(renderWidth, renderHeight);
// Set the frame duration to an appropriate value (i.e. 30 frames per second for
video).
mutableVideoComposition.frameDuration = CMTimeMake(1,30);
```

Exporting the Composition and Saving it to the Camera Roll

The final step in this process involves exporting the entire composition into a single video file and saving that video to the camera roll. You use an `AVAssetExportSession` object to create the new video file and you pass to it your desired URL for the output file. You can then use the `ALAssetsLibrary` class to save the resulting video file to the Camera Roll.

```
// Create a static date formatter so we only have to initialize it once.
static NSDateFormatter *kDateFormatter;
if (!kDateFormatter) {
    kDateFormatter = [[NSDateFormatter alloc] init];
    kDateFormatter.dateStyle = NSDateFormatterMediumStyle;
    kDateFormatter.timeStyle = NSDateFormatterShortStyle;
}
// Create the export session with the composition and set the preset to the highest
 quality.
AVAssetExportSession *exporter = [[AVAssetExportSession alloc]
initWithAsset:mutableComposition presetName:AVAssetExportPresetHighestQuality];
// Set the desired output URL for the file created by the export process.
exporter.outputURL = [[[[NSFileManager defaultManager]
URLForDirectory:NSDocumentDirectory inDomain:NSUserDomainMask appropriateForURL:nil
 create:@YES error:nil] URLByAppendingPathComponent:[kDateFormatter
stringFromDate:[NSDate date]]]
URLByAppendingPathExtension:CFBridgingRelease(UTTypeCopyPreferredTagWithClass((CFStringRef)AVFileTypeQuickTimeMovie,
 kUTTagClassFilenameExtension))];
// Set the output file type to be a QuickTime movie.
exporter.outputFileType = AVFileTypeQuickTimeMovie;
exporter.shouldOptimizeForNetworkUse = YES;
exporter.videoComposition = mutableVideoComposition;
```

```
// Asynchronously export the composition to a video file and save this file to the
 camera roll once export completes.
[exporter exportAsynchronouslyWithCompletionHandler:^{
    dispatch_async(dispatch_get_main_queue(), ^{
        if (exporter.status == AVAssetExportSessionStatusCompleted) {
            ALAssetsLibrary *assetsLibrary = [[ALAssetsLibrary alloc] init];
            if ([assetsLibrary
videoAtPathIsCompatibleWithSavedPhotosAlbum:exporter.outputURL]) {
              [assetsLibrary writeVideoAtPathToSavedPhotosAlbum:exporter.outputURL
 completionBlock:NULL];
            }
        }
    });
}];
```

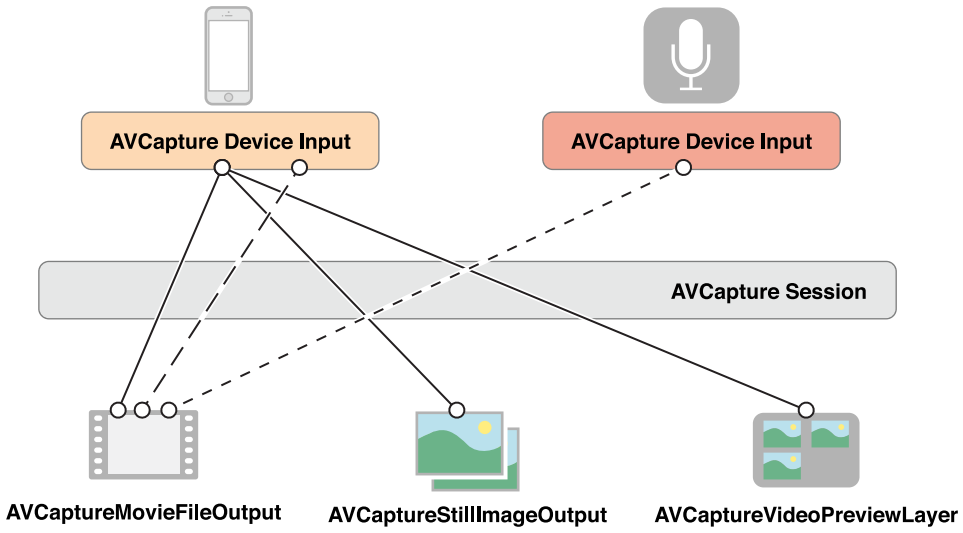
Still and Video Media Capture

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

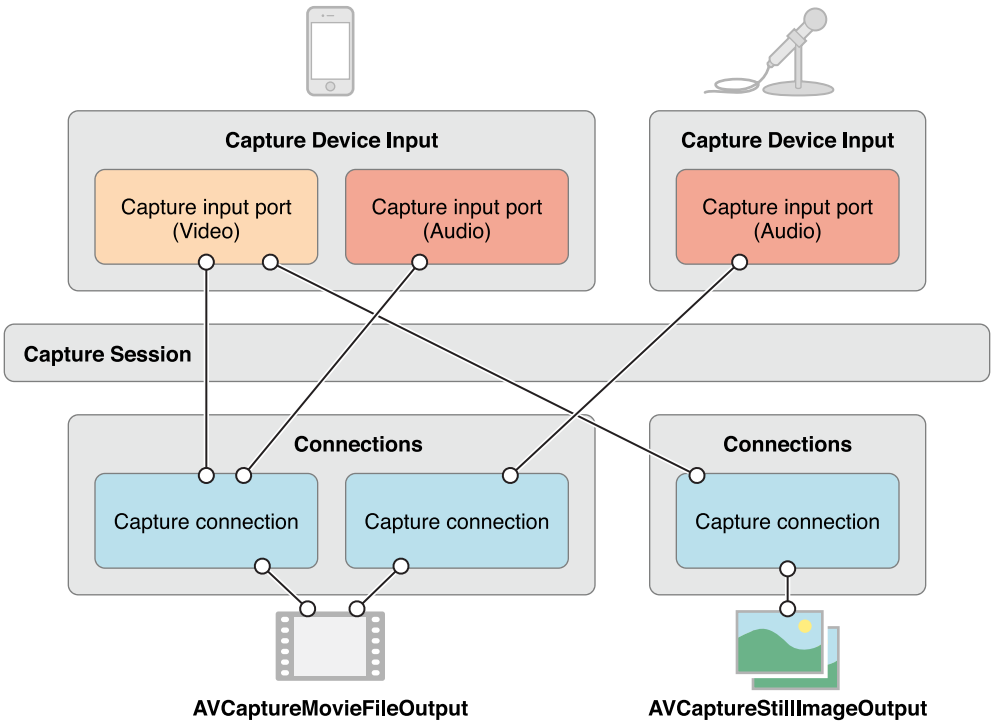
You can configure multiple inputs and outputs, coordinated by a single session:



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 and 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 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.

Note: Media capture does not support simultaneous capture of both the front-facing and back-facing cameras on iOS devices.

Use a Capture Session to Coordinate Data Flow

An `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 the data flow 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 Wi-Fi 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.

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

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

If you need to adjust session parameters at a more granular level than is possible with a preset, or you'd like to make changes to a running session, you surround your changes with the `beginConfiguration` and `commitConfiguration` methods. The `beginConfiguration` and `commitConfiguration` methods ensure that devices changes occur as a group, minimizing visibility or inconsistency of state. After calling `beginConfiguration`, you can add or remove outputs, alter the `sessionPreset` property, 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 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. Additionally, both the `running` and `interrupted` properties are key-value observing compliant and the notifications are posted on the main thread.

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, for example, two video inputs—one for the front-facing the camera, one for the back-facing camera—and one audio input for the microphone.

You can find out which capture devices are currently available using the `AVCaptureDevice` class methods `devices` and `devicesWithMediaType:`. And, if necessary, you can find out what features an iPhone, iPad, or iPod offers (see Device Capture Settings (page 56)). The list of available devices may change, though. Current input devices may become unavailable (if they're used by another application), and new input 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 an input device to a capture session using a capture input (see Use Capture Inputs to Add a Capture Device to a Session (page 61)).

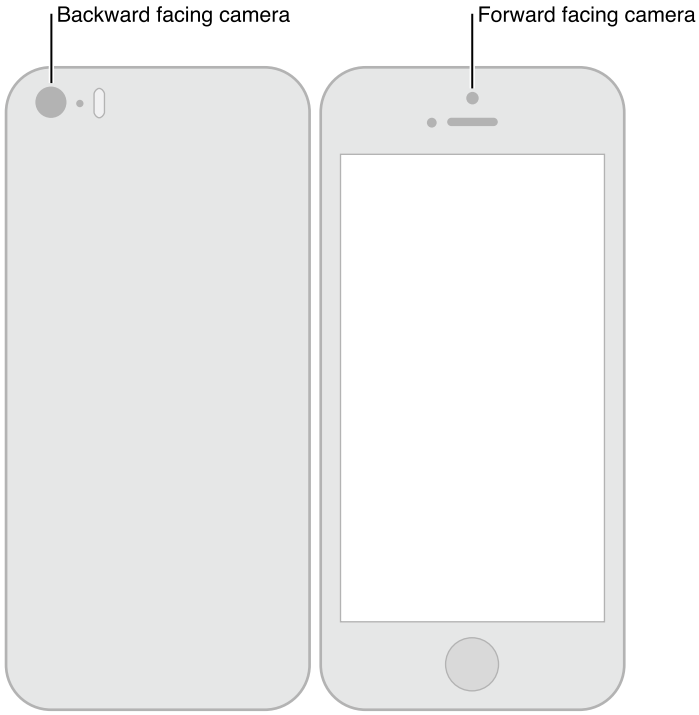
Device Characteristics

You can ask a device about its different characteristics. You can also 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 being tested), 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.

Figure 4-1 shows the positions of the back-facing (`AVCaptureDevicePositionBack`) and front-facing (`AVCaptureDevicePositionFront`) cameras.

Note: Media capture does not support simultaneous capture of both the front-facing and back-facing cameras on iOS devices.

Figure 4-1 iOS device front and back facing camera positions



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.

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 60).

> **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 position 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 autofocuses 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:

- AVCaptureExposureModeContinuousAutoExposure: The device automatically adjusts the exposure level as needed.
- `AVCaptureExposureModeLocked`: The exposure level is fixed at its current level.

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 dependent on the ambient light conditions.

You use `hasFlash` to determine whether a device has a flash. If that method returns `YES`, you then use the `isFlashModeSupported:` method, passing the desired mode to determine whether a device supports a given flash mode, then set the mode using the `flashMode` property.

Torch Mode

In torch mode, the flash is continuously enabled 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 automatically 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.

Video Stabilization

Cinematic vdeo stabilization is available for connections that operate on video, depending on the specific device hardware. Even so, not all source formats and video resolutions are supported.

Enabling cinematic video stabilization may also introduce additional latency into the video capture pipeline. To detect when video stabilization is in use, use the `videoStabilizationEnabled` property. The `enablesVideoStabilizationWhenAvailable` property allows an application to automatically enable video stabilization if it is supported by the camera. By default automatic stabilization is disabled due to the above limitations.

White Balance

There are two white balance modes:

- `AVCaptureWhiteBalanceModeLocked`: The white balance mode is fixed.
- `AVCaptureWhiteBalanceModeContinuousAutoWhiteBalance`: The camera continuously adjusts 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.

Setting Device Orientation

You set the desired orientation on a `AVCaptureConnection` to specify how you want the images oriented in the `AVCaptureOutput` (`AVCaptureMovieFileOutput`, `AVCaptureStillImageOutput` and `AVCaptureVideoDataOutput`) for the connection.

Use the `AVCaptureConnectionsupportsVideoOrientation` property to determine whether the device supports changing the orientation of the video, and the `videoOrientation` property to specify how you want the images oriented in the output port. Listing 4-1 shows how to set the orientation for a `AVCaptureConnection` to `AVCaptureVideoOrientationLandscapeLeft`:

Listing 4-1 Setting the orientation of a capture connection

```
AVCaptureConnection *captureConnection = <#A capture connection#>;
if ([captureConnection isVideoOrientationSupported])
{
    AVCaptureVideoOrientation orientation = AVCaptureVideoOrientationLandscapeLeft;
     [captureConnection setVideoOrientation:orientation];
}
```

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 hold the device lock only if you need the 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 users to switch between input devices—for example, switching from using the front-facing to to the back-facing 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;
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 52) 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, for example, to create your own custom view layer
- `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 required while the session is running.

```
AVCaptureSession *captureSession = <#Get a capture session#>;
AVCaptureMovieFileOutput *movieOutput = <#Create and configure a movie output#>;
if ([captureSession canAddOutput:movieOutput]) {
    [captureSession addOutput:movieOutput];
}
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 a recording, or its 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 is typically AAC. The actual values vary by device.

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, because 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 album. It should also check for any errors that might have occurred.

Ensuring That 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`key in the user info dictionary of the error, 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—`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;
NSMutableArray *newMetadataArray = nil;
if (existingMetadataArray) {
    newMetadataArray = [existingMetadataArray mutableCopy];
}
else {
    newMetadataArray = [[NSMutableArray 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 setting 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 can use the queue to modify the priority given to delivering and processing the video frames. See *SquareCam* for a sample implementation.

The frames are presented in the delegate method, `captureOutput:didOutputSampleBuffer:fromConnection:`, as instances of the `CMSampleBufferRef` opaque type (see Representations of Media (page 108)). 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 formats are returned by the `availableVideoCVPixelFormatTypes` property , and the `availableVideoCodecTypes` property returns the supported values. Both Core Graphics and OpenGL work well with the `BGRA` format:

```
AVCaptureVideoDataOutput *videoDataOutput = [AVCaptureVideoDataOutput new];
NSDictionary *newSettings =
                @{ (NSString *)kCVPixelBufferPixelFormatTypeKey :
@(kCVPixelFormatType_32BGRA) };
videoDataOutput.videoSettings = newSettings;


 // discard if the data output queue is blocked (as we process the still image
[videoDataOutput setAlwaysDiscardsLateVideoFrames:YES];)


// create a serial dispatch queue used for the sample buffer delegate as well as
when a still image is captured
// a serial dispatch queue must be used to guarantee that video frames will be
delivered in order
// see the header doc for setSampleBufferDelegate:queue: for more information
videoDataOutputQueue = dispatch_queue_create("VideoDataOutputQueue",
DISPATCH_QUEUE_SERIAL);
[videoDataOutput setSampleBufferDelegate:self queue:videoDataOutputQueue];


AVCaptureSession *captureSession = <#The Capture Session#>;


if ( [captureSession canAddOutput:videoDataOutput] )
     [captureSession addOutput:videoDataOutput];
```

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 stops delivering frames, not only to your delegate but also to other outputs such as a preview layer.

You can use the capture video data output's `minFrameDuration` property to be sure 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 make sure 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 and 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 that 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 well as the device.

Pixel and Encoding Formats

Different devices support different image formats. You can find out what pixel and codec types are supported by a device using `availableImageDataCVPixelFormatTypes` and `availableImageDataCodecTypes` respectively. Each method returns an array of the supported values for the specific device. You set the `outputSettings` dictionary to specify the image format you want, for example:

```
AVCaptureStillImageOutput *stillImageOutput = [[AVCaptureStillImageOutput alloc]
init];
NSDictionary *outputSettings = @{ AVVideoCodecKey : AVVideoCodecJPEG};
[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 recompressing 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` opaque type 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 if you want, but note the optimization for JPEG images discussed in Pixel and Encoding Formats (page 67).

```
[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.

Using the `AVCaptureVideoDataOutput` class provides the client application with the ability to access the video pixels before they are presented to the user.

Unlike a capture output, a video preview layer maintains a strong reference to 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, you can test for device support for video mirroring by querying the `supportsVideoMirroring` property. You can set the `videoMirrored` property as required, although when the `automaticallyAdjustsVideoMirroring` property is set to `YES` (the default), the mirroring value is automatically set based on the configuration of the session.

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 for the possibility that the preview may be mirrored. See the sample code project *AVCam for iOS* for an implementation of this functionality.

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. If an appropriate device can not be located, then the `deviceInputWithDevice:error:` method will return an error by reference.

```
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 Video 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];
[session addOutput:output];
output.videoSettings =
                @{ (NSString *)kCVPixelBufferPixelFormatTypeKey :
@(kCVPixelFormatType_32BGRA) };
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 `CMSampleBuffer` opaque types, so you need to convert from the `CMSampleBuffer` opaque type to a `UIImage` object. The function for this operation is shown in Converting CMSampleBuffer to a UIImage Object (page 109).

```
- (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 should ensure that the camera has permission to record according to the user's preferences.

```
NSString *mediaType = AVMediaTypeVideo;


[AVCaptureDevice requestAccessForMediaType:mediaType completionHandler:^(BOOL
granted) {
    if (granted)
    {
        //Granted access to mediaType
        [self setDeviceAuthorized:YES];
    }
    else
    {
        //Not granted access to mediaType
        dispatch_async(dispatch_get_main_queue(), ^{
        [[[UIAlertView alloc] initWithTitle:@"AVCam!"
                                    message:@"AVCam doesn't have permission to use
 Camera, please change privacy settings"
                                    delegate:self
                           cancelButtonTitle:@"OK"
                           otherButtonTitles:nil] show];
                [self setDeviceAuthorized:NO];
        });
    }
}];
```

If the camera session is configured and the user has approved access to the camera (and if required, the microphone), send a `startRunning` message to start the recording.

Important: The `startRunning` method is a blocking call which can take some time, therefore you should perform session setup on a serial queue so that the main queue isn't blocked (which keeps the UI responsive). See *AVCam for iOS* for the canonical implementation example.

```
[session startRunning];
```

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

High Frame Rate Video Capture

iOS 7.0 introduces high frame rate video capture support (also referred to as "SloMo" video) on selected hardware. The full AV Foundation framework supports high frame rate content.

You determine the capture capabilities of a device using the `AVCaptureDeviceFormat` class. This class has methods that return the supported media types, frame rates, field of view, maximum zoom factor, whether video stabilization is supported, and more.

- Capture supports full 720p (1280 x 720 pixels) resolution at 60 frames per second (fps) including video stabilization and droppable P-frames (a feature of H264 encoded movies, which allow the movies to play back smoothly even on slower and older hardware.)
- Playback has enhanced audio support for slow and fast playback, allowing the time pitch of the audio can be preserved at slower or faster speeds.
- Editing has full support for scaled edits in mutable compositions.
- Export provides two options when supporting 60 fps movies. The variable frame rate, slow or fast motion, can be preserved, or the movie and be converted to an arbitrary slower frame rate such as 30 frames per second.

The *SloPoke* sample code demonstrates the AV Foundation support for fast video capture, determining whether hardware supports high frame rate video capture, playback using various rates and time pitch algorithms, and editing (including setting time scales for portions of a composition).

Playback

An instance of `AVPlayer` manages most of the playback speed automatically by setting the `setRate:` method value. The value is used as a multiplier for the playback speed. A value of 1.0 causes normal playback, 0.5 plays back at half speed, 5.0 plays back five times faster than normal, and so on.

The `AVPlayerItem` object supports the `audioTimePitchAlgorithm` property. This property allows you to specify how audio is played when the movie is played at various frame rates using the `Time Pitch Algorithm Settings` constants.

The following table shows the supported time pitch algorithms, the quality, whether the algorithm causes the audio to snap to specific frame rates, and the frame rate range that each algorithm supports.

Time pitch algorithm	Quality	Snaps to specific frame rate	Rate range
`AVAudioTimePitch-AlgorithmLowQuality-ZeroLatency`	Low quality, suitable for fast-forward, rewind, or low quality voice.	`YES`	0.5, 0.666667, 0.8, 1.0, 1.25, 1.5, 2.0 rates.
`AVAudioTimePitch-AlgorithmTimeDomain`	Modest quality, less expensive computationally, suitable for voice.	`NO`	0.5–2x rates.
`AVAudioTimePitch-AlgorithmSpectral`	Highest quality, most expensive computationally, preserves the pitch of the original item.	`NO`	1/32–32 rates.
`AVAudioTimePitch-AlgorithmVarispeed`	High-quality playback with no pitch correction.	`NO`	1/32–32 rates.

Editing

When editing, you use the `AVMutableComposition` class to build temporal edits.

- Create a new `AVMutableComposition` instance using the `composition` class method.
- Insert your video asset using the `insertTimeRange:ofAsset:atTime:error:` method.
- Set the time scale of a portion of the composition using `scaleTimeRange:toDuration:`

Export

Exporting 60 fps video uses the `AVAssetExportSession` class to export an asset. The content can be exported using two techniques:

- Use the `AVAssetExportPresetPassthrough` preset to avoid reencoding the movie. It retimes the media with the sections of the media tagged as section 60 fps, section slowed down, or section sped up.
- Use a constant frame rate export for maximum playback compatibility. Set the `frameDuration` property of the video composition to 30 fps. You can also specify the time pitch by using setting the export session's `audioTimePitchAlgorithm` property.

Recording

You capture high frame rate video using the `AVCaptureMovieFileOutput` class, which automatically supports high frame rate recording. It will automatically select the correct H264 pitch level and bit rate.

To do custom recording, you must use the `AVAssetWriter` class, which requires some additional setup.

```
assetWriterInput.expectsMediaDataInRealTime=YES;
```

This setting ensures that the capture can keep up with the incoming data.

Export

To read and write audiovisual assets, you must use the export APIs provided by the AV Foundation framework. The `AVAssetExportSession` class provides an interface for simple exporting needs, such as modifying the file format or trimming the length of an asset (see Trimming and Transcoding a Movie (page 18)). For more in-depth exporting needs, use the `AVAssetReader` and `AVAssetWriter` classes.

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, use an `AVAssetWriter` object.

> **Note:** The asset reader and writer classes are not intended to be used for real-time processing. In fact, an asset reader cannot even be used for reading from a real-time source like an HTTP live stream. However, if you are using an asset writer with a real-time data source, such as an `AVCaptureOutput` object, set the `expectsMediaDataInRealTime` property of your asset writer's inputs to YES. Setting this property to YES for a non-real-time data source will result in your files not being interleaved properly.

Reading an Asset

Each `AVAssetReader` object can be associated only with a single asset at a time, but this asset may contain multiple tracks. For this reason, you must assign concrete subclasses of the `AVAssetReaderOutput` class to your asset reader before you begin reading in order to configure how the media data is read. There are three concrete subclasses of the `AVAssetReaderOutput` base class that you can use for your asset reading needs: `AVAssetReaderTrackOutput`, `AVAssetReaderAudioMixOutput`, and `AVAssetReaderVideoCompositionOutput`.

Creating the Asset Reader

All you need to initialize an `AVAssetReader` object is the asset that you want to read.

```
NSError *outError;
AVAsset *someAsset = <#AVAsset that you want to read#>;
```

```
AVAssetReader *assetReader = [AVAssetReader assetReaderWithAsset:someAsset
error:&outError];
BOOL success = (assetReader != nil);
```

Note: Always check that the asset reader returned to you is non-`nil` to ensure that the asset reader was initialized successfully. Otherwise, the error parameter (`outError` in the previous example) will contain the relevant error information.

Setting Up the Asset Reader Outputs

After you have created your asset reader, set up at least one output to receive the media data being read. When setting up your outputs, be sure to set the `alwaysCopiesSampleData` property to `NO`. In this way, you reap the benefits of performance improvements. In all of the examples within this chapter, this property could and should be set to `NO`.

If you want only to read media data from one or more tracks and potentially convert that data to a different format, use the `AVAssetReaderTrackOutput` class, using a single track output object for each `AVAssetTrack` object that you want to read from your asset. To decompress an audio track to Linear PCM with an asset reader, you set up your track output as follows:

```
AVAsset *localAsset = assetReader.asset;
// Get the audio track to read.
AVAssetTrack *audioTrack = [[localAsset tracksWithMediaType:AVMediaTypeAudio]
objectAtIndex:0];
// Decompression settings for Linear PCM
NSDictionary *decompressionAudioSettings = @{ AVFormatIDKey : [NSNumber
numberWithUnsignedInt:kAudioFormatLinearPCM] };
// Create the output with the audio track and decompression settings.
AVAssetReaderOutput *trackOutput = [AVAssetReaderTrackOutput
assetReaderTrackOutputWithTrack:audioTrack
outputSettings:decompressionAudioSettings];
// Add the output to the reader if possible.
if ([assetReader canAddOutput:trackOutput])
    [assetReader addOutput:trackOutput];
```

> **Note:** To read the media data from a specific asset track in the format in which it was stored, pass `nil` to the `outputSettings` parameter.

You use the `AVAssetReaderAudioMixOutput` and `AVAssetReaderVideoCompositionOutput` classes to read media data that has been mixed or composited together using an `AVAudioMix` object or `AVVideoComposition` object, respectively. Typically, these outputs are used when your asset reader is reading from an `AVComposition` object.

With a single audio mix output, you can read multiple audio tracks from your asset that have been mixed together using an `AVAudioMix` object. To specify how the audio tracks are mixed, assign the mix to the `AVAssetReaderAudioMixOutput` object after initialization. The following code displays how to create an audio mix output with all of the audio tracks from your asset, decompress the audio tracks to Linear PCM, and assign an audio mix object to the output. For details on how to configure an audio mix, see Editing (page 35).

```
AVAudioMix *audioMix = <#An AVAudioMix that specifies how the audio tracks from
the AVAsset are mixed#>;

// Assumes that assetReader was initialized with an AVComposition object.

AVComposition *composition = (AVComposition *)assetReader.asset;

// Get the audio tracks to read.

NSArray *audioTracks = [composition tracksWithMediaType:AVMediaTypeAudio];

// Get the decompression settings for Linear PCM.

NSDictionary *decompressionAudioSettings = @{ AVFormatIDKey : [NSNumber
numberWithUnsignedInt:kAudioFormatLinearPCM] };

// Create the audio mix output with the audio tracks and decompression setttings.

AVAssetReaderOutput *audioMixOutput = [AVAssetReaderAudioMixOutput
assetReaderAudioMixOutputWithAudioTracks:audioTracks
audioSettings:decompressionAudioSettings];

// Associate the audio mix used to mix the audio tracks being read with the output.

audioMixOutput.audioMix = audioMix;

// Add the output to the reader if possible.

if ([assetReader canAddOutput:audioMixOutput])

    [assetReader addOutput:audioMixOutput];
```

> **Note:** Passing `nil` for the `audioSettings` parameter tells the asset reader to return samples in a convenient uncompressed format. The same is true for the `AVAssetReaderVideoCompositionOutput` class.

The video composition output behaves in much the same way: You can read multiple video tracks from your asset that have been composited together using an `AVVideoComposition` object. To read the media data from multiple composited video tracks and decompress it to ARGB, set up your output as follows:

```
AVVideoComposition *videoComposition = <#An AVVideoComposition that specifies how
 the video tracks from the AVAsset are composited#>;

// Assumes assetReader was initialized with an AVComposition.

AVComposition *composition = (AVComposition *)assetReader.asset;

// Get the video tracks to read.

NSArray *videoTracks = [composition tracksWithMediaType:AVMediaTypeVideo];

// Decompression settings for ARGB.

NSDictionary *decompressionVideoSettings = @{ (id)kCVPixelBufferPixelFormatTypeKey
 : [NSNumber numberWithUnsignedInt:kCVPixelFormatType_32ARGB],
(id)kCVPixelBufferIOSurfacePropertiesKey : [NSDictionary dictionary] };

// Create the video composition output with the video tracks and decompression
setttings.

AVAssetReaderOutput *videoCompositionOutput = [AVAssetReaderVideoCompositionOutput
 assetReaderVideoCompositionOutputWithVideoTracks:videoTracks
videoSettings:decompressionVideoSettings];

// Associate the video composition used to composite the video tracks being read
with the output.

videoCompositionOutput.videoComposition = videoComposition;

// Add the output to the reader if possible.

if ([assetReader canAddOutput:videoCompositionOutput])

    [assetReader addOutput:videoCompositionOutput];
```

Reading the Asset's Media Data

To start reading after setting up all of the outputs you need, call the `startReading` method on your asset reader. Next, retrieve the media data individually from each output using the `copyNextSampleBuffer` method. To start up an asset reader with a single output and read all of its media samples, do the following:

```
// Start the asset reader up.

[self.assetReader startReading];
```

```
BOOL done = NO;
while (!done)
{
  // Copy the next sample buffer from the reader output.
  CMSampleBufferRef sampleBuffer = [self.assetReaderOutput copyNextSampleBuffer];
  if (sampleBuffer)
  {
    // Do something with sampleBuffer here.
    CFRelease(sampleBuffer);
    sampleBuffer = NULL;
  }
  else
  {
    // Find out why the asset reader output couldn't copy another sample buffer.
    if (self.assetReader.status == AVAssetReaderStatusFailed)
    {
      NSError *failureError = self.assetReader.error;
      // Handle the error here.
    }
    else
    {
      // The asset reader output has read all of its samples.
      done = YES;
    }
  }
}
```

Writing an Asset

The `AVAssetWriter` class to write media data from multiple sources to a single file of a specified file format. You don't need to associate your asset writer object with a specific asset, but you must use a separate asset writer for each output file that you want to create. Because an asset writer can write media data from multiple sources, you must create an `AVAssetWriterInput` object for each individual track that you want to write to

the output file. Each `AVAssetWriterInput` object expects to receive data in the form of `CMSampleBufferRef` objects, but if you want to append `CVPixelBufferRef` objects to your asset writer input, use the `AVAssetWriterInputPixelBufferAdaptor` class.

Creating the Asset Writer

To create an asset writer, specify the URL for the output file and the desired file type. The following code displays how to initialize an asset writer to create a QuickTime movie:

```
NSError *outError;
NSURL *outputURL = <#NSURL object representing the URL where you want to save the
 video#>;
AVAssetWriter *assetWriter = [AVAssetWriter assetWriterWithURL:outputURL

fileType:AVFileTypeQuickTimeMovie
                                                          error:&outError];
BOOL success = (assetWriter != nil);
```

Setting Up the Asset Writer Inputs

For your asset writer to be able to write media data, you must set up at least one asset writer input. For example, if your source of media data is already vending media samples as `CMSampleBufferRef` objects, just use the `AVAssetWriterInput` class. To set up an asset writer input that compresses audio media data to 128 kbps AAC and connect it to your asset writer, do the following:

```
// Configure the channel layout as stereo.
AudioChannelLayout stereoChannelLayout = {
    .mChannelLayoutTag = kAudioChannelLayoutTag_Stereo,
    .mChannelBitmap = 0,
    .mNumberChannelDescriptions = 0
};

// Convert the channel layout object to an NSData object.
NSData *channelLayoutAsData = [NSData dataWithBytes:&stereoChannelLayout
length:offsetof(AudioChannelLayout, mChannelDescriptions)];

// Get the compression settings for 128 kbps AAC.
NSDictionary *compressionAudioSettings = @{
```

```
    AVFormatIDKey         : [NSNumber numberWithUnsignedInt:kAudioFormatMPEG4AAC],
    AVEncoderBitRateKey   : [NSNumber numberWithInteger:128000],
    AVSampleRateKey       : [NSNumber numberWithInteger:44100],
    AVChannelLayoutKey    : channelLayoutAsData,
    AVNumberOfChannelsKey : [NSNumber numberWithUnsignedInteger:2]
};


// Create the asset writer input with the compression settings and specify the
media type as audio.
AVAssetWriterInput *assetWriterInput = [AVAssetWriterInput
assetWriterInputWithMediaType:AVMediaTypeAudio
outputSettings:compressionAudioSettings];
// Add the input to the writer if possible.
if ([assetWriter canAddInput:assetWriterInput])
    [assetWriter addInput:assetWriterInput];
```

> **Note:** If you want the media data to be written in the format in which it was stored, pass `nil` in the `outputSettings` parameter. Pass `nil` only if the asset writer was initialized with a `fileType` of `AVFileTypeQuickTimeMovie`.

Your asset writer input can optionally include some metadata or specify a different transform for a particular track using the `metadata` and `transform` properties respectively. For an asset writer input whose data source is a video track, you can maintain the video's original transform in the output file by doing the following:

```
AVAsset *videoAsset = <#AVAsset with at least one video track#>;
AVAssetTrack *videoAssetTrack = [[videoAsset tracksWithMediaType:AVMediaTypeVideo]
 objectAtIndex:0];
assetWriterInput.transform = videoAssetTrack.preferredTransform;
```

> **Note:** Set the `metadata` and `transform` properties before you begin writing with your asset writer for them to take effect.

When writing media data to the output file, sometimes you may want to allocate pixel buffers. To do so, use the `AVAssetWriterInputPixelBufferAdaptor` class. For greatest efficiency, instead of adding pixel buffers that were allocated using a separate pool, use the pixel buffer pool provided by the pixel buffer adaptor. The following code creates a pixel buffer object working in the RGB domain that will use `CGImage` objects to create its pixel buffers.

```
NSDictionary *pixelBufferAttributes = @{
     kCVPixelBufferCGImageCompatibilityKey : [NSNumber numberWithBool:YES],
    kCVPixelBufferCGBitmapContextCompatibilityKey : [NSNumber numberWithBool:YES],
     kCVPixelBufferPixelFormatTypeKey : [NSNumber
numberWithInt:kCVPixelFormatType_32ARGB]
};
AVAssetWriterInputPixelBufferAdaptor *inputPixelBufferAdaptor =
[AVAssetWriterInputPixelBufferAdaptor
assetWriterInputPixelBufferAdaptorWithAssetWriterInput:self.assetWriterInput
sourcePixelBufferAttributes:pixelBufferAttributes];
```

> **Note:** All `AVAssetWriterInputPixelBufferAdaptor` objects must be connected to a single asset writer input. That asset writer input must accept media data of type `AVMediaTypeVideo`.

Writing Media Data

When you have configured all of the inputs needed for your asset writer, you are ready to begin writing media data. As you did with the asset reader, initiate the writing process with a call to the `startWriting` method. You then need to start a sample-writing session with a call to the `startSessionAtSourceTime:` method. All writing done by an asset writer has to occur within one of these sessions and the time range of each session defines the time range of media data included from within the source. For example, if your source is an asset reader that is supplying media data read from an `AVAsset` object and you don't want to include media data from the first half of the asset, you would do the following:

```
CMTime halfAssetDuration = CMTimeMultiplyByFloat64(self.asset.duration, 0.5);
[self.assetWriter startSessionAtSourceTime:halfAssetDuration];
//Implementation continues.
```

Normally, to end a writing session you must call the `endSessionAtSourceTime:` method. However, if your writing session goes right up to the end of your file, you can end the writing session simply by calling the `finishWriting` method. To start up an asset writer with a single input and write all of its media data, do the following:

```
// Prepare the asset writer for writing.
[self.assetWriter startWriting];
// Start a sample-writing session.
[self.assetWriter startSessionAtSourceTime:kCMTimeZero];
// Specify the block to execute when the asset writer is ready for media data and
 the queue to call it on.
[self.assetWriterInput requestMediaDataWhenReadyOnQueue:myInputSerialQueue
usingBlock:^{
     while ([self.assetWriterInput isReadyForMoreMediaData])
     {
          // Get the next sample buffer.
         CMSampleBufferRef nextSampleBuffer = [self copyNextSampleBufferToWrite];
          if (nextSampleBuffer)
          {
               // If it exists, append the next sample buffer to the output file.
               [self.assetWriterInput appendSampleBuffer:nextSampleBuffer];
               CFRelease(nextSampleBuffer);
               nextSampleBuffer = nil;
          }
          else
          {
              // Assume that lack of a next sample buffer means the sample buffer
 source is out of samples and mark the input as finished.
               [self.assetWriterInput markAsFinished];
               break;
          }
     }
}];
```

The `copyNextSampleBufferToWrite` method in the code above is simply a stub. The location of this stub is where you would need to insert some logic to return `CMSampleBufferRef` objects representing the media data that you want to write. One possible source of sample buffers is an asset reader output.

Reencoding Assets

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 an `AVAssetExportSession` object. For example, you can 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. The first step in this process is just to set up your asset reader outputs and asset writer inputs as desired. After your asset reader and writer are fully configured, you start up both of them with calls to the `startReading` and `startWriting` methods, respectively. The following code snippet displays how to use a single asset writer input to write media data supplied by a single asset reader output:

```
NSString *serializationQueueDescription = [NSString stringWithFormat:@"%@
serialization queue", self];


// Create a serialization queue for reading and writing.

dispatch_queue_t serializationQueue =
dispatch_queue_create([serializationQueueDescription UTF8String], NULL);


// Specify the block to execute when the asset writer is ready for media data and
 the queue to call it on.

[self.assetWriterInput requestMediaDataWhenReadyOnQueue:serializationQueue
usingBlock:^{

     while ([self.assetWriterInput isReadyForMoreMediaData])

     {

          // Get the asset reader output's next sample buffer.

          CMSampleBufferRef sampleBuffer = [self.assetReaderOutput
copyNextSampleBuffer];

          if (sampleBuffer != NULL)

          {

               // If it exists, append this sample buffer to the output file.

               BOOL success = [self.assetWriterInput
appendSampleBuffer:sampleBuffer];

               CFRelease(sampleBuffer);

               sampleBuffer = NULL;

               // Check for errors that may have occurred when appending the new
sample buffer.

            if (!success && self.assetWriter.status == AVAssetWriterStatusFailed)

               {

                    NSError *failureError = self.assetWriter.error;
```

```
                    //Handle the error.
                }
            }
            else
            {
                // If the next sample buffer doesn't exist, find out why the asset
 reader output couldn't vend another one.
                if (self.assetReader.status == AVAssetReaderStatusFailed)
                {
                    NSError *failureError = self.assetReader.error;
                    //Handle the error here.
                }
                else
                {
                   // The asset reader output must have vended all of its samples.
 Mark the input as finished.
                    [self.assetWriterInput markAsFinished];
                    break;
                }
            }
        }
}];
```

Putting It All Together: Using an Asset Reader and Writer in Tandem to Reencode an Asset

This brief code example illustrates how to use an asset reader and writer to reencode the first video and audio track of an asset into a new file. It shows how to:

- Use serialization queues to handle the asynchronous nature of reading and writing audiovisual data
- Initialize an asset reader and configure two asset reader outputs, one for audio and one for video
- Initialize an asset writer and configure two asset writer inputs, one for audio and one for video
- Use an asset reader to asynchronously supply media data to an asset writer through two different output/input combinations
- Use a dispatch group to be notified of completion of the reencoding process

- Allow a user to cancel the reencoding process once it has begun

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

Handling the Initial Setup

Before you create your asset reader and writer and configure their outputs and inputs, you need to handle some initial setup. The first part of this setup involves creating three separate serialization queues to coordinate the reading and writing process.

```
NSString *serializationQueueDescription = [NSString stringWithFormat:@"%@
serialization queue", self];


// Create the main serialization queue.
self.mainSerializationQueue = dispatch_queue_create([serializationQueueDescription
 UTF8String], NULL);
NSString *rwAudioSerializationQueueDescription = [NSString stringWithFormat:@"%@
rw audio serialization queue", self];


// Create the serialization queue to use for reading and writing the audio data.
self.rwAudioSerializationQueue =
dispatch_queue_create([rwAudioSerializationQueueDescription UTF8String], NULL);
NSString *rwVideoSerializationQueueDescription = [NSString stringWithFormat:@"%@
rw video serialization queue", self];


// Create the serialization queue to use for reading and writing the video data.
self.rwVideoSerializationQueue =
dispatch_queue_create([rwVideoSerializationQueueDescription UTF8String], NULL);
```

The main serialization queue is used to coordinate the starting and stopping of the asset reader and writer (perhaps due to cancellation) and the other two serialization queues are used to serialize the reading and writing by each output/input combination with a potential cancellation.

Now that you have some serialization queues, load the tracks of your asset and begin the reencoding process.

```
self.asset = <#AVAsset that you want to reencode#>;
self.cancelled = NO;
```

```
self.outputURL = <#NSURL representing desired output URL for file generated by
asset writer#>;

// Asynchronously load the tracks of the asset you want to read.

[self.asset loadValuesAsynchronouslyForKeys:@[@"tracks"] completionHandler:^{

     // Once the tracks have finished loading, dispatch the work to the main
serialization queue.

     dispatch_async(self.mainSerializationQueue, ^{

         // Due to asynchronous nature, check to see if user has already cancelled.

          if (self.cancelled)

               return;

          BOOL success = YES;

          NSError *localError = nil;

          // Check for success of loading the assets tracks.

          success = ([self.asset statusOfValueForKey:@"tracks" error:&localError]
 == AVKeyValueStatusLoaded);

          if (success)

          {

              // If the tracks loaded successfully, make sure that no file exists
 at the output path for the asset writer.

               NSFileManager *fm = [NSFileManager defaultManager];

               NSString *localOutputPath = [self.outputURL path];

               if ([fm fileExistsAtPath:localOutputPath])

                    success = [fm removeItemAtPath:localOutputPath
error:&localError];

          }

          if (success)

               success = [self setupAssetReaderAndAssetWriter:&localError];

          if (success)

               success = [self startAssetReaderAndWriter:&localError];

          if (!success)

               [self readingAndWritingDidFinishSuccessfully:success
withError:localError];

     });

}];
```

When the track loading process finishes, whether successfully or not, the rest of the work is dispatched to the main serialization queue to ensure that all of this work is serialized with a potential cancellation. Now all that's left is to implement the cancellation process and the three custom methods at the end of the previous code listing.

Initializing the Asset Reader and Writer

The custom `setupAssetReaderAndAssetWriter:` method initializes the reader and writer and configures two output/input combinations, one for an audio track and one for a video track. In this example, the audio is decompressed to Linear PCM using the asset reader and compressed back to 128 kbps AAC using the asset writer. The video is decompressed to YUV using the asset reader and compressed to H.264 using the asset writer.

```
- (BOOL)setupAssetReaderAndAssetWriter:(NSError **)outError
{
    // Create and initialize the asset reader.
    self.assetReader = [[AVAssetReader alloc] initWithAsset:self.asset
error:outError];
    BOOL success = (self.assetReader != nil);
    if (success)
    {
        // If the asset reader was successfully initialized, do the same for the
 asset writer.
        self.assetWriter = [[AVAssetWriter alloc] initWithURL:self.outputURL
fileType:AVFileTypeQuickTimeMovie error:outError];
        success = (self.assetWriter != nil);
    }

    if (success)
    {
        // If the reader and writer were successfully initialized, grab the audio
 and video asset tracks that will be used.
        AVAssetTrack *assetAudioTrack = nil, *assetVideoTrack = nil;
        NSArray *audioTracks = [self.asset tracksWithMediaType:AVMediaTypeAudio];
        if ([audioTracks count] > 0)
            assetAudioTrack = [audioTracks objectAtIndex:0];
        NSArray *videoTracks = [self.asset tracksWithMediaType:AVMediaTypeVideo];
        if ([videoTracks count] > 0)
```

```
            assetVideoTrack = [videoTracks objectAtIndex:0];


        if (assetAudioTrack)
        {
            // If there is an audio track to read, set the decompression settings
 to Linear PCM and create the asset reader output.
            NSDictionary *decompressionAudioSettings = @{ AVFormatIDKey :
[NSNumber numberWithUnsignedInt:kAudioFormatLinearPCM] };
            self.assetReaderAudioOutput = [AVAssetReaderTrackOutput
assetReaderTrackOutputWithTrack:assetAudioTrack
outputSettings:decompressionAudioSettings];
            [self.assetReader addOutput:self.assetReaderAudioOutput];
            // Then, set the compression settings to 128kbps AAC and create the
 asset writer input.
            AudioChannelLayout stereoChannelLayout = {
                .mChannelLayoutTag = kAudioChannelLayoutTag_Stereo,
                .mChannelBitmap = 0,
                .mNumberChannelDescriptions = 0
            };
            NSData *channelLayoutAsData = [NSData
dataWithBytes:&stereoChannelLayout length:offsetof(AudioChannelLayout,
mChannelDescriptions)];
            NSDictionary *compressionAudioSettings = @{
                AVFormatIDKey         : [NSNumber
numberWithUnsignedInt:kAudioFormatMPEG4AAC],
                AVEncoderBitRateKey   : [NSNumber numberWithInteger:128000],
                AVSampleRateKey       : [NSNumber numberWithInteger:44100],
                AVChannelLayoutKey    : channelLayoutAsData,
                AVNumberOfChannelsKey : [NSNumber numberWithUnsignedInteger:2]
            };
            self.assetWriterAudioInput = [AVAssetWriterInput
assetWriterInputWithMediaType:[assetAudioTrack mediaType]
outputSettings:compressionAudioSettings];
            [self.assetWriter addInput:self.assetWriterAudioInput];
        }


        if (assetVideoTrack)
        {
```

```
            // If there is a video track to read, set the decompression settings
 for YUV and create the asset reader output.
            NSDictionary *decompressionVideoSettings = @{
                 (id)kCVPixelBufferPixelFormatTypeKey     : [NSNumber
numberWithUnsignedInt:kCVPixelFormatType_422YpCbCr8],
                 (id)kCVPixelBufferIOSurfacePropertiesKey : [NSDictionary
dictionary]
            };
            self.assetReaderVideoOutput = [AVAssetReaderTrackOutput
assetReaderTrackOutputWithTrack:assetVideoTrack
outputSettings:decompressionVideoSettings];
            [self.assetReader addOutput:self.assetReaderVideoOutput];
            CMFormatDescriptionRef formatDescription = NULL;
            // Grab the video format descriptions from the video track and grab
 the first one if it exists.
            NSArray *videoFormatDescriptions = [assetVideoTrack
formatDescriptions];
            if ([videoFormatDescriptions count] > 0)
                 formatDescription = (__bridge
CMFormatDescriptionRef)[formatDescriptions objectAtIndex:0];
            CGSize trackDimensions = {
                 .width = 0.0,
                 .height = 0.0,
            };
            // If the video track had a format description, grab the track
dimensions from there. Otherwise, grab them direcly from the track itself.
            if (formatDescription)
                 trackDimensions =
CMVideoFormatDescriptionGetPresentationDimensions(formatDescription, false, false);
            else
                 trackDimensions = [assetVideoTrack naturalSize];
            NSDictionary *compressionSettings = nil;
            // If the video track had a format description, attempt to grab the
 clean aperture settings and pixel aspect ratio used by the video.
            if (formatDescription)
            {
                 NSDictionary *cleanAperture = nil;
                 NSDictionary *pixelAspectRatio = nil;
```

```
                    CFDictionaryRef cleanApertureFromCMFormatDescription =
CMFormatDescriptionGetExtension(formatDescription,
kCMFormatDescriptionExtension_CleanAperture);
                    if (cleanApertureFromCMFormatDescription)
                    {
                         cleanAperture = @{
                              AVVideoCleanApertureWidthKey            :
(id)CFDictionaryGetValue(cleanApertureFromCMFormatDescription,
kCMFormatDescriptionKey_CleanApertureWidth),
                              AVVideoCleanApertureHeightKey           :
(id)CFDictionaryGetValue(cleanApertureFromCMFormatDescription,
kCMFormatDescriptionKey_CleanApertureHeight),
                              AVVideoCleanApertureHorizontalOffsetKey :
(id)CFDictionaryGetValue(cleanApertureFromCMFormatDescription,
kCMFormatDescriptionKey_CleanApertureHorizontalOffset),
                              AVVideoCleanApertureVerticalOffsetKey   :
(id)CFDictionaryGetValue(cleanApertureFromCMFormatDescription,
kCMFormatDescriptionKey_CleanApertureVerticalOffset)
                         };
                    }
                    CFDictionaryRef pixelAspectRatioFromCMFormatDescription =
CMFormatDescriptionGetExtension(formatDescription,
kCMFormatDescriptionExtension_PixelAspectRatio);
                    if (pixelAspectRatioFromCMFormatDescription)
                    {
                         pixelAspectRatio = @{
                              AVVideoPixelAspectRatioHorizontalSpacingKey :
(id)CFDictionaryGetValue(pixelAspectRatioFromCMFormatDescription,
kCMFormatDescriptionKey_PixelAspectRatioHorizontalSpacing),
                              AVVideoPixelAspectRatioVerticalSpacingKey   :
(id)CFDictionaryGetValue(pixelAspectRatioFromCMFormatDescription,
kCMFormatDescriptionKey_PixelAspectRatioVerticalSpacing)
                         };
                    }
                    // Add whichever settings we could grab from the format
description to the compression settings dictionary.
                    if (cleanAperture || pixelAspectRatio)
                    {
                         NSMutableDictionary *mutableCompressionSettings =
[NSMutableDictionary dictionary];
                         if (cleanAperture)
```

```
                                [mutableCompressionSettings setObject:cleanAperture
 forKey:AVVideoCleanApertureKey];
                            if (pixelAspectRatio)
                              [mutableCompressionSettings setObject:pixelAspectRatio
 forKey:AVVideoPixelAspectRatioKey];
                            compressionSettings = mutableCompressionSettings;
                        }
                    }
                    // Create the video settings dictionary for H.264.
                    NSMutableDictionary *videoSettings = (NSMutableDictionary *) @{
                        AVVideoCodecKey  : AVVideoCodecH264,
                        AVVideoWidthKey  : [NSNumber
numberWithDouble:trackDimensions.width],
                        AVVideoHeightKey : [NSNumber
numberWithDouble:trackDimensions.height]
                    };
                    // Put the compression settings into the video settings dictionary
 if we were able to grab them.
                    if (compressionSettings)
                        [videoSettings setObject:compressionSettings
forKey:AVVideoCompressionPropertiesKey];
                    // Create the asset writer input and add it to the asset writer.
                    self.assetWriterVideoInput = [AVAssetWriterInput
assetWriterInputWithMediaType:[videoTrack mediaType] outputSettings:videoSettings];
                    [self.assetWriter addInput:self.assetWriterVideoInput];
                }
            }
            return success;
}
```

Reencoding the Asset

Provided that the asset reader and writer are successfully initialized and configured, the `startAssetReaderAndWriter:` method described in Handling the Initial Setup (page 88) is called. This method is where the actual reading and writing of the asset takes place.

```
- (BOOL)startAssetReaderAndWriter:(NSError **)outError
{
```

```
    BOOL success = YES;
    // Attempt to start the asset reader.
    success = [self.assetReader startReading];
    if (!success)
        *outError = [self.assetReader error];
    if (success)
    {
        // If the reader started successfully, attempt to start the asset writer.
        success = [self.assetWriter startWriting];
        if (!success)
            *outError = [self.assetWriter error];
    }


    if (success)
    {
        // If the asset reader and writer both started successfully, create the
 dispatch group where the reencoding will take place and start a sample-writing
session.
        self.dispatchGroup = dispatch_group_create();
        [self.assetWriter startSessionAtSourceTime:kCMTimeZero];
        self.audioFinished = NO;
        self.videoFinished = NO;


        if (self.assetWriterAudioInput)
        {
            // If there is audio to reencode, enter the dispatch group before
beginning the work.
            dispatch_group_enter(self.dispatchGroup);
            // Specify the block to execute when the asset writer is ready for
 audio media data, and specify the queue to call it on.
            [self.assetWriterAudioInput
requestMediaDataWhenReadyOnQueue:self.rwAudioSerializationQueue usingBlock:^{
                // Because the block is called asynchronously, check to see
whether its task is complete.
                if (self.audioFinished)
                    return;
                BOOL completedOrFailed = NO;
```

```
                    // If the task isn't complete yet, make sure that the input
is actually ready for more media data.
                    while ([self.assetWriterAudioInput isReadyForMoreMediaData]
&& !completedOrFailed)
                    {
                        // Get the next audio sample buffer, and append it to the
 output file.
                        CMSampleBufferRef sampleBuffer =
[self.assetReaderAudioOutput copyNextSampleBuffer];
                        if (sampleBuffer != NULL)
                        {
                            BOOL success = [self.assetWriterAudioInput
appendSampleBuffer:sampleBuffer];
                            CFRelease(sampleBuffer);
                            sampleBuffer = NULL;
                            completedOrFailed = !success;
                        }
                        else
                        {
                            completedOrFailed = YES;
                        }
                    }
                    if (completedOrFailed)
                    {
                        // Mark the input as finished, but only if we haven't
already done so, and then leave the dispatch group (since the audio work has
finished).
                        BOOL oldFinished = self.audioFinished;
                        self.audioFinished = YES;
                        if (oldFinished == NO)
                        {
                            [self.assetWriterAudioInput markAsFinished];
                        }
                        dispatch_group_leave(self.dispatchGroup);
                    }
                }];
            }
```

```
          if (self.assetWriterVideoInput)
          {
               // If we had video to reencode, enter the dispatch group before
beginning the work.
               dispatch_group_enter(self.dispatchGroup);
               // Specify the block to execute when the asset writer is ready for
 video media data, and specify the queue to call it on.
               [self.assetWriterVideoInput
requestMediaDataWhenReadyOnQueue:self.rwVideoSerializationQueue usingBlock:^{
                    // Because the block is called asynchronously, check to see
whether its task is complete.
                    if (self.videoFinished)
                         return;
                    BOOL completedOrFailed = NO;
                    // If the task isn't complete yet, make sure that the input
is actually ready for more media data.
                    while ([self.assetWriterVideoInput isReadyForMoreMediaData]
&& !completedOrFailed)
                    {
                        // Get the next video sample buffer, and append it to the
 output file.
                         CMSampleBufferRef sampleBuffer =
[self.assetReaderVideoOutput copyNextSampleBuffer];
                         if (sampleBuffer != NULL)
                         {
                              BOOL success = [self.assetWriterVideoInput
appendSampleBuffer:sampleBuffer];
                              CFRelease(sampleBuffer);
                              sampleBuffer = NULL;
                              completedOrFailed = !success;
                         }
                         else
                         {
                              completedOrFailed = YES;
                         }
                    }
                    if (completedOrFailed)
```

```
                    {
                        // Mark the input as finished, but only if we haven't
already done so, and then leave the dispatch group (since the video work has
finished).
                        BOOL oldFinished = self.videoFinished;
                        self.videoFinished = YES;
                        if (oldFinished == NO)
                        {
                            [self.assetWriterVideoInput markAsFinished];
                        }
                        dispatch_group_leave(self.dispatchGroup);
                    }
                }];
            }
            // Set up the notification that the dispatch group will send when the
audio and video work have both finished.
            dispatch_group_notify(self.dispatchGroup, self.mainSerializationQueue,
^{
                BOOL finalSuccess = YES;
                NSError *finalError = nil;
                // Check to see if the work has finished due to cancellation.
                if (self.cancelled)
                {
                    // If so, cancel the reader and writer.
                    [self.assetReader cancelReading];
                    [self.assetWriter cancelWriting];
                }
                else
                {
                   // If cancellation didn't occur, first make sure that the asset
 reader didn't fail.
                    if ([self.assetReader status] == AVAssetReaderStatusFailed)
                    {
                        finalSuccess = NO;
                        finalError = [self.assetReader error];
                    }
```

```
                    // If the asset reader didn't fail, attempt to stop the asset
 writer and check for any errors.
                    if (finalSuccess)
                    {
                         finalSuccess = [self.assetWriter finishWriting];
                         if (!finalSuccess)
                              finalError = [self.assetWriter error];
                    }
               }
              // Call the method to handle completion, and pass in the appropriate
 parameters to indicate whether reencoding was successful.
               [self readingAndWritingDidFinishSuccessfully:finalSuccess
withError:finalError];
          });
     }
     // Return success here to indicate whether the asset reader and writer were
started successfully.
     return success;
}
```

During reencoding, the audio and video tracks are asynchronously handled on individual serialization queues to increase the overall performance of the process, but both queues are contained within the same dispatch group. By placing the work for each track within the same dispatch group, the group can send a notification when all of the work is done and the success of the reencoding process can be determined.

Handling Completion

To handle the completion of the reading and writing process, the `readingAndWritingDidFinishSuccessfully:` method is called—with parameters indicating whether or not the reencoding completed successfully. If the process didn't finish successfully, the asset reader and writer are both canceled and any UI related tasks are dispatched to the main queue.

```
- (void)readingAndWritingDidFinishSuccessfully:(BOOL)success withError:(NSError
*)error
{
     if (!success)
     {
```

```
        // If the reencoding process failed, we need to cancel the asset reader
 and writer.
        [self.assetReader cancelReading];
        [self.assetWriter cancelWriting];
        dispatch_async(dispatch_get_main_queue(), ^{
            // Handle any UI tasks here related to failure.
        });
    }
    else
    {
        // Reencoding was successful, reset booleans.
        self.cancelled = NO;
        self.videoFinished = NO;
        self.audioFinished = NO;
        dispatch_async(dispatch_get_main_queue(), ^{
            // Handle any UI tasks here related to success.
        });
    }
}
```

Handling Cancellation

Using multiple serialization queues, you can allow the user of your app to cancel the reencoding process with ease. On the main serialization queue, messages are asynchronously sent to each of the asset reencoding serialization queues to cancel their reading and writing. When these two serialization queues complete their cancellation, the dispatch group sends a notification to the main serialization queue where the `cancelled` property is set to `YES`. You might associate the `cancel` method from the following code listing with a button on your UI.

```
- (void)cancel
{
    // Handle cancellation asynchronously, but serialize it with the main queue.
    dispatch_async(self.mainSerializationQueue, ^{
        // If we had audio data to reencode, we need to cancel the audio work.
        if (self.assetWriterAudioInput)
        {
```

```
            // Handle cancellation asynchronously again, but this time serialize
 it with the audio queue.
             dispatch_async(self.rwAudioSerializationQueue, ^{
                 // Update the Boolean property indicating the task is complete
 and mark the input as finished if it hasn't already been marked as such.
                 BOOL oldFinished = self.audioFinished;
                 self.audioFinished = YES;
                 if (oldFinished == NO)
                 {
                     [self.assetWriterAudioInput markAsFinished];
                 }
                 // Leave the dispatch group since the audio work is finished
now.
                 dispatch_group_leave(self.dispatchGroup);
             });
         }


         if (self.assetWriterVideoInput)
         {
             // Handle cancellation asynchronously again, but this time serialize
 it with the video queue.
             dispatch_async(self.rwVideoSerializationQueue, ^{
                 // Update the Boolean property indicating the task is complete
 and mark the input as finished if it hasn't already been marked as such.
                 BOOL oldFinished = self.videoFinished;
                 self.videoFinished = YES;
                 if (oldFinished == NO)
                 {
                     [self.assetWriterVideoInput markAsFinished];
                 }
                 // Leave the dispatch group, since the video work is finished
 now.
                 dispatch_group_leave(self.dispatchGroup);
             });
         }
         // Set the cancelled Boolean property to YES to cancel any work on the
main queue as well.
```

```
            self.cancelled = YES;
        });
}
```

Asset Output Settings Assistant

The `AVOutputSettingsAssistant` class aids in creating output-settings dictionaries for an asset reader or writer. This makes setup much simpler, especially for high frame rate H264 movies that have a number of specific presets. Listing 5-1 shows an example that uses the output settings assistant to use the settings assistant.

Listing 5-1 AVOutputSettingsAssistant sample

```
AVOutputSettingsAssistant *outputSettingsAssistant = [AVOutputSettingsAssistant
outputSettingsAssistantWithPreset:<some preset>];

CMFormatDescriptionRef audioFormat = [self getAudioFormat];


if (audioFormat != NULL)

    [outputSettingsAssistant
setSourceAudioFormat:(CMAudioFormatDescriptionRef)audioFormat];


CMFormatDescriptionRef videoFormat = [self getVideoFormat];


if (videoFormat != NULL)

    [outputSettingsAssistant
setSourceVideoFormat:(CMVideoFormatDescriptionRef)videoFormat];


CMTime assetMinVideoFrameDuration = [self getMinFrameDuration];

CMTime averageFrameDuration = [self getAvgFrameDuration]


[outputSettingsAssistant setSourceVideoAverageFrameDuration:averageFrameDuration];

[outputSettingsAssistant setSourceVideoMinFrameDuration:assetMinVideoFrameDuration];


AVAssetWriter *assetWriter = [AVAssetWriter assetWriterWithURL:<some URL>
fileType:[outputSettingsAssistant outputFileType] error:NULL];

AVAssetWriterInput *audioInput = [AVAssetWriterInput
assetWriterInputWithMediaType:AVMediaTypeAudio
outputSettings:[outputSettingsAssistant audioSettings] sourceFormatHint:audioFormat];
```

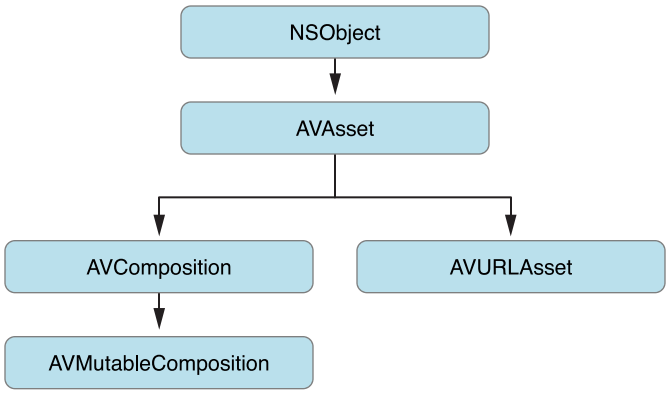
```
AVAssetWriterInput *videoInput = [AVAssetWriterInput
assetWriterInputWithMediaType:AVMediaTypeVideo
outputSettings:[outputSettingsAssistant videoSettings] sourceFormatHint:videoFormat];
```

Time and Media Representations

Time-based audiovisual 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.

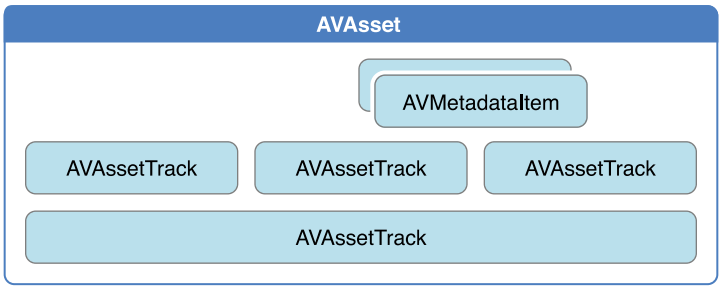
Representation of Assets

`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 9)), 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 12)).



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 `CMFormatDescription` objects (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, because this is a multiple of several commonly used frame rates: 24 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` structure can represent nonnumeric 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 timescale). There are several functions for time-based arithmetic and for comparing 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 in which a `CMTime` structure can, for example, represent a time that is invalid. To test whether a `CMTime` is valid, or a nonnumeric 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` structure with `kCMTimeInvalid`.

Representing CMTime as an Object

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

Epochs

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

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

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, to determine 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` structure can be invalid, or zero—or indefinite (if one of the `CMTime` structures is indefinite. If you need to test whether a `CMTimeRange` structure 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` structure with `kCMTimeRangeInvalid`.

Representing a CMTimeRange Structure as an Object

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

Representations of Media

Video data and its associated metadata are 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 CMSampleBuffer to a UIImage Object (page 109).

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 CMSampleBuffer to a UIImage Object

The following code 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.

```
// Create a UIImage from sample buffer data
- (UIImage *) imageFromSampleBuffer:(CMSampleBufferRef) sampleBuffer
{
    // Get a CMSampleBuffer's Core Video image buffer for the media data
    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
    void *baseAddress = CVPixelBufferGetBaseAddress(imageBuffer);

    // 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
    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();

    // Create a bitmap graphics context with the sample buffer data
    CGContextRef context = CGBitmapContextCreate(baseAddress, width, height, 8,
      bytesPerRow, colorSpace, kCGBitmapByteOrder32Little |
kCGImageAlphaPremultipliedFirst);
    // Create a Quartz image from the pixel data in the bitmap graphics context
    CGImageRef quartzImage = CGBitmapContextCreateImage(context);
    // Unlock the pixel buffer
    CVPixelBufferUnlockBaseAddress(imageBuffer,0);

    // Free up the context and color space
    CGContextRelease(context);
    CGColorSpaceRelease(colorSpace);

    // Create an image object from the Quartz image
    UIImage *image = [UIImage imageWithCGImage:quartzImage];

    // Release the Quartz image
    CGImageRelease(quartzImage);

    return (image);
}
```

Document Revision History

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

Date	Notes
2015-01-12	Updated About AV Foundation to add names to AV Foundation stacks on iOS and OS X. Added links to the camera examples.
2014-03-10	Updated for iOS 7.0 and OS X v10.9.
2013-10-22	Clarified use of CMSampleBufferGetImageBuffer.
2013-08-08	Added a new chapter centered around the editing API provided by the AV Foundation framework.
2011-10-12	Updated for iOS5 to include references to release notes.
2011-04-28	First release for OS X v10.7.
2010-09-08	Expanded Playback and Metadata chapters.
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