What's New with Screen Recording and Live Broadcast

Session 606

Johnny Trenh, Software Engineer
Alexander Subbotin, Software Engineer
Screen Recording

Record app visuals and audio

Record microphone input

Share recordings
Live Broadcast

Broadcast app visuals and audio

Xcode Templates
Details

- HD quality
- Low-performance impact
- Minimal power usage
- Privacy safeguards
- User consent prompt
ReplayKit 2
ReplayKit 2

In-App Screen Capture
ReplayKit 2

- In-App Screen Capture
- iOS Screen Record and Broadcast
ReplayKit 2

In-App Screen Capture

iOS Screen Record and Broadcast

Broadcast Pairing
ReplayKit 2

- In-App Screen Capture
- iOS Screen Record and Broadcast
- Broadcast Pairing
- Fast Camera Switching
In-App Screen Capture

Johnny Trenh, Software Engineer
Screen Recording

Application

- RPScreenRecorder
- RPPreviewViewController

System

- Replay Daemon
- Movie
- Preview and Share Extension
In-App Screen Capture

Application

RPScreenRecorder
In-App Screen Capture

Direct access to audio and video samples
More control and flexibility
HD quality
Minimal power usage
Privacy safeguards
Simple API
// Capture API

func startCapture(handler captureHandler: ((CMSampleBuffer, RPSampleBufferType, Error?) -> Void)?, completionHandler: ((Error?) -> Void)? = nil)

func stopCapture(handler: ((Error?) -> Void)? = nil)
// Initiating Capture

func didPressCaptureButton() {
    let sharedRecorder = RPScreenRecorder.shared()
    sharedRecorder.startCapture(handler: { (cmSampleBuffer, rpSampleType, error) in
        // Handle Samples Passed Back from ReplayKit
    })(error) { (error) in
        // Update UI
    }
}
func didPressCaptureButton() {
    let sharedRecorder = RPScreenRecorder.shared()
    sharedRecorder.startCapture(handler: { (cmSampleBuffer, rpSampleType, error) in
        switch rpSampleType {
            case RPSampleBufferTypeVideo:
                self.videoInput.appendSampleBuffer(samples)
            case RPSampleBufferTypeAudio:
                self.audioInput.appendSampleBuffer(samples)
            case RPSampleBufferTypeMic:
                self.micInput.appendSampleBuffer(samples)
            default:
                println("sample has no matching type")
        }
    }) { (error) in
        updateCaptureButton(didCompleteError:error)
    }
}
func didPressCaptureButton() {
    let sharedRecorder = RPScreenRecorder.shared()
    sharedRecorder.startCapture(handler: { (cmSampleBuffer, rpSampleType, error) in
        switch rpSampleType {
        case RPSampleBufferTypeVideo:
            self.videoInput.appendSampleBuffer(samples)
        case RPSampleBufferTypeAudio:
            self.audioInput.appendSampleBuffer(samples)
        case RPSampleBufferTypeMic:
            self.micInput.appendSampleBuffer(samples)
        default:
            println("sample has no matching type")
        }
    }) { (error) in
        updateCaptureButton(didCompleteError:error)
    }
}
// Completion Handler Block

func didPressCaptureButton() {
    let sharedRecorder = RPScreenRecorder.shared()
    sharedRecorder.startCapture(handler: { (cmSampleBuffer, rpSampleType, error) in
        switch rpSampleType {
        case RPSampleBufferTypeVideo:
            self.videoInput.appendSampleBuffer(samples)
        case RPSampleBufferTypeAudio:
            self.audioInput.appendSampleBuffer(samples)
        case RPSampleBufferTypeMic:
            self.micInput.appendSampleBuffer(samples)
        default:
            println("sample has no matching type")
        }
    }) { (error) in
        updateCaptureButton(didCompleteError:error)
    }
}
// Completion Handler Block

func didPressCaptureButton() {
    let sharedRecorder = RPScreenRecorder.shared()
    sharedRecorder.startCapture(handler: { (cmSampleBuffer, rpSampleType, error) in
        switch rpSampleType {
        case RPSampleBufferTypeVideo:
            self.videoInput.appendSampleBuffer(samples)
        case RPSampleBufferTypeAudio:
            self.audioInput.appendSampleBuffer(samples)
        case RPSampleBufferTypeMic:
            self.micInput.appendSampleBuffer(samples)
        default:
            println("sample has no matching type")
        }
    }) { (error) in
        updateCaptureButton(didCompleteError:error)
    }
}
Screen Capture Architecture

Application

RPScreenRecorder

System

Replay Daemon

Audio/Video Samples
Create and Manage Video
Create and Manage Video
Custom Video Editor
Custom Video Editor
Share Application Screen

Columns, bar, and pie charts compare values in a single category, such as the number of products sold by each salesperson. The charts show each category's value as a percentage of the whole.

**SHARED EXPENSES**

Calculate how much each person in your group owes. Enter the details of your transactions in the Expenses table, then use the chart in the Person column of the Expenses table. You can add or remove rows and columns as necessary. As you add expenses, the amount each person owes is calculated and shown in the chart.
Share Application Screen

Columns, bar, and pie charts compare values in a single category, such as the number of products sold by each supplier. The charts show each category's value as a percentage of the whole.

**Shared Expenses**

To calculate the share each person has to pay, add or remove rows and columns as necessary. You can add or remove rows and columns as necessary. As you add expenses, the amount each person owes is calculated and shown on the chart.
Share Application Screen

Columns, bar, and pie charts compare values in a single category, such as the number of products sold by each salesperson. The charts show each category's value as a percentage of the whole.

SHARED EXPENSES

Calculate how much each person in your group owes. Enter the details of your transactions in the Expenses table, then sign the name of the person in your group in the Person column of the Expenses table. You can add or remove rows and columns as necessary. As you add expenses, the accumulated person row is calculated and shown in the chart.
iOS Screen Recording and Broadcast
Control Center
Handling Interruptions

In-App Recording can be interrupted by iOS Screen Record and Broadcast

Recordings will be discarded if interrupted by iOS Screen Record and Broadcast

Application will get delegate call with appropriate RPError
Live Broadcast

Alexander Subbotin, Software Engineer
Live Broadcast

Broadcast app visuals and audio

iOS and tvOS

Can include microphone, camera feed

Content is secure
Live Broadcast

Client App

ReplayKit

Setup UI Extension

Upload Extension

A/V Samples

A/V Stream

Broadcast App

A/V Stream

Live Broadcast
Live Broadcast

Client App → ReplayKit → Setup UI Extension → Upload Extension → A/V Stream → Broadcast App

A/V Samples → Upload Extension → A/V Stream → Broadcast App
Live Broadcast

- **Client App**
  - Communicates with **ReplayKit**
- **ReplayKit**
  - Exchanges **A/V Samples** with **Setup UI Extension**
  - Serves as a bridge to **Upload Extension**
- **Setup UI Extension**
  - Connects to **Upload Extension**
- **Upload Extension**
  - Exchanges **A/V Stream** with **Broadcaster App**
  - Facilitates the delivery of content to a global audience
Broadcast API
App Client

RPBroadcastActivityViewController
- Allow user to select broadcaster

RPBroadcastController
- Manage broadcast (start, stop, pause, resume)

RPBroadcastControllerDelegate
- Handle broadcast events

User Enters Broadcast Information
User Selects Broadcaster
User Initiates Broadcast
Broadcast Starts
Broadcast API

App extensions

Broadcast Setup UI extension
• Account sign-in, broadcast title

Broadcast Upload extension
• Encode samples, upload to service

Each runs in its own process

Installed with broadcaster’s app
Broadcast API

Xcode templates
Broadcast SetupUI Extension

UI to login, get broadcast name and other
Uploads name and icon of the app
Provides broadcast URL back to client app
Broadcast SetupUI Extension

NSExtensionContext(RPBroadcastExtension)
• loadBroadcastingApplicationInfo
• completeRequest(withBroadcast)
// Get name and icon for client application and provide to the broadcast service

class BroadcastSetupViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        extensionContext?.loadBroadcastingApplicationInfo(completion: {
            (bundleID, displayName, appIcon) in
            broadcastSession.setAppInfo(bundleID, displayName, appIcon)
        })
    }
}
Get name and icon for client application and provide to the broadcast service.

```swift
class BroadcastSetupViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        extensionContext?.loadBroadcastingApplicationInfo(completion: {
            (bundleID, displayName, appIcon) in
            broadcastSession.setAppInfo(bundleID, displayName, appIcon)
        })
    }
}
```
// Complete setup extension request with broadcastURL and setupInfo

class BroadcastSetupViewController: UIViewController {

    func done() {
        let broadcastURL = URL(string: "http://myCompany.com/broadcast/streamID")

        let setupInfo: [String : NSCoding & NSObjectProtocol] = [
            "broadcastName": "example" as NSCoding & NSObjectProtocol
        ]

        extensionContext?.completeRequest(withBroadcast: broadcastURL!, setupInfo: setupInfo)
    }
}

Complete setup extension request with broadcastURL and setupInfo

class BroadcastSetupViewController: UIViewController {
    func done() {
        let broadcastURL = URL(string: "http://myCompany.com/broadcast/streamID")

        let setupInfo: [String : NSCoding & NSObjectProtocol] = [
            "broadcastName": "example" as NSCoding & NSObjectProtocol
        ]

        extensionContext?.completeRequest(withBroadcast: broadcastURL!, setupInfo: setupInfo)
    }
}
// You should always provide option to cancel broadcast

class BroadcastSetupViewController: UIViewController {
    func cancel() {
        let error = NSError(domain: "broadcast", code: -1, userInfo: nil)
        extensionContext?.cancelRequest(withError: error)
    }
}

// You should always provide option to cancel broadcast

class BroadcastSetupViewController: UIViewController {
    func cancel() {
        let error = NSError(domain: "broadcast", code: -1, userInfo: nil)
        extensionContext?.cancelRequest(withError: error)
    }
}


Broadcast Upload Extension

- Receives audio and video samples
- Encodes and uploads stream to broadcaster
// SampleHandler created by Xcode templates for Upload Extension

class SampleHandler: RPBroadcastSampleHandler {

    override func broadcastStarted(withSetupInfo setupInfo: [String : NSObject]?) {
        // User has requested to start the broadcast
    }

    override func broadcastPaused() {
        // User has requested to pause the broadcast. Samples will stop being delivered.
    }

    override func broadcastResumed() {
        // User has requested to resume the broadcast. Samples delivery will resume.
    }

    override func broadcastFinished() {
        // User has requested to finish the broadcast
    }

    override func processSampleBuffer(_ sampleBuffer: CMSampleBuffer,
                                      with sampleBufferType: RPSampleBufferType) {
        // Handle the sample buffer here
    }
}
// Override broadcastStarted to prepare to receive media samples
override func broadcastStarted(withSetupInfo setupInfo: [String : NSObject]?) {
    if (setupInfo != nil) {
        session.broadcastDescription.name = setupInfo["name"]
    } else {
        session.broadcastDescription.iOSScreenBroadcast = true
    }
}
// Override broadcastStarted to prepare to receive media samples
override func broadcastStarted(withSetupInfo setupInfo: [String : NSObject]?) {
    if (setupInfo != nil) {
        session.broadcastDescription.name = setupInfo["name"]
    } else {
        session.broadcastDescription.iOSScreenBroadcast = true
    }
}
Broadcast Upload Extension

processSampleBuffer

- Video (screen)
- Audio (app)
- Audio (mic)
Both audio and video samples are handled by processSampleBuffer routine.

```swift
override func processSampleBuffer(_ sampleBuffer: CMSampleBuffer,
                        with sampleBufferType: RPSampleBufferType) {

    switch sampleBufferType {
        case RPSampleBufferType.video:
            var imageBuffer: CVImageBuffer = CMSampleBufferGetImageBuffer(sampleBuffer)!
            var pts = CMSampleBufferGetPresentationTimeStamp(sampleBuffer) as CMTime
            VTCompressionSessionEncodeFrame(session, imageBuffer, pts,
                kCMTimeInvalid, nil, nil, nil)
            break
        case RPSampleBufferType.audioApp:
            // Handle audio sample buffer for app audio
            break
        case RPSampleBufferType.audioMic:
            // Handle audio sample buffer for mic audio
            break
    }
}
```
/ Both audio and video samples are handled by processSampleBuffer routine

override func processSampleBuffer(_ sampleBuffer: CMSampleBuffer,
                                 with sampleBufferType: RPSampleBufferType) {

switch sampleBufferType {

    case RPSampleBufferType.video:
        var imageBuffer:CVImageBuffer = CMSampleBufferGetImageBuffer(sampleBuffer)!
        var pts = CMSampleBufferGetPresentationTimeStamp(sampleBuffer) as CMTime
        VTCompressionSessionEncodeFrame(session, imageBuffer, pts,
                                        kCMT imeInvalid, nil, nil, nil)
        break

    case RPSampleBufferType.audioApp:
    // Handle audio sample buffer for app audio
        break

    case RPSampleBufferType.audioMic:
    // Handle audio sample buffer for mic audio
        break

    }
}
Live Broadcast

Client App → ReplayKit → Setup UI Extension → Upload Extension → Broadcaster App → A/V Stream

A/V Samples
Service Information

Information provided by the service during broadcast

• Dictionary
• KVO observable

Can use to communicate service data back to user

• Likes, viewer count, and chat

```swift
class SampleHandler: RPBroadcastSampleHandler {
    func updateViewersCount(_ count: UInt) {
        updateServiceInfo(["count": (count)])
    }
}
```
Broadcast Pairing
Broadcast Pairing Flow

- Budget App
- Conference App

Broadcast Extension
Broadcast Pairing Flow

Application

RPBroadcastActivity
ViewController

Budget App

Conference App

Broadcast Extension

Ton
The budget looks great!

Jlb
I'm really excited about our financial outlook.
Broadcast Pairing Flow

Application
RPBroadcastActivity ViewController

Budget App

Broadcast Extension

Conference App

NEW
Broadcast Pairing Flow

1. Application
2. RPBroadcastActivity ViewController
3. User Enters Broadcast Information
4. Broadcast Extension

Budget App

Conference App
Broadcast Pairing Flow

1. Application
2. RPBroadcastActivity ViewController
3. User Enters Broadcast Information
4. Broadcast Starts
5. Broadcast Extension
6. Conference App
7. Budget App

- User enters broadcast information, which triggers the start of a broadcast.
- The broadcast extension is initiated, enabling communication between the applications.
- The Conference App displays a message from the Budget App: "The budget looks great!"
- A comment from User 2: "I'm really excited about our financial outlook."
// Broadcast Pairing API

class func load(withPreferredExtension preferredExtension: String?, handler: @escaping (RPBroadcastActivityViewController?, Error?) -> Void)
func didPressBroadcastPairButton () {

RPBroadcastActivityViewController.load(withPreferredExtension:"com.conferenceApp.broadcastExtension") { (broadcastAVC, error) in
    broadcastAVC?.delegate = self
    self.present(broadcastAVC, animated: true, completion: nil)
}
}
func didPressBroadcastPairButton () {

    RPBroadcastActivityViewController.load(withPreferredExtension: "com.conferenceApp.broadcastExtension") { (broadcastAVC, error) in
        broadcastAVC?.delegate = self
        self.present(broadcastAVC, animated: true, completion: nil)
    }
}

Initiating Broadcast Pairing
func didPressBroadcastPairButton () {

RPBroadcastActivityViewController.load(withPreferredExtension: "com.conferenceApp.broadcastExtension") { (broadcastAVC, error) in
    broadcastAVC?.delegate = self
    self.present(broadcastAVC, animated: true, completion: nil)
}
}
Broadcast Pairing

Application provides extension bundleID

User approves extension
Fast Camera Switching
Fast Camera Switching

Front Camera and Back Camera switching

Camera preview view available in RPScreenRecorder

Subclass of UIView

Developer is responsible for UI elements for fast switching
Fast Camera Switching

RPScreenRecorder.cameraPosition

```swift
var cameraPosition: RPCameraPosition
```

RPCameraPosition

```swift
public enum RPCameraPosition: Int {
    case front
    case back
}
```
func showPreviewView() {
    let sharedRecorder = RPScreenRecorder.shared()
    let cameraView = sharedRecorder.cameraPreviewView
    self.view.addSubview(cameraView!)
}
func didPressCameraSwitch() {
    let sharedRecorder = RPScreenRecorder.shared()
    if (sharedRecorder.cameraPosition == RPCameraPosition.back) {
        sharedRecorder.cameraPosition = RPCameraPosition.front
    } else {
        sharedRecorder.cameraPosition = RPCameraPosition.back
    }
}
func didPressCameraSwitch() {
    let sharedRecorder = RPScreenRecorder.shared()
    if (sharedRecorder.cameraPosition == RPCameraPosition.back) {
        sharedRecorder.cameraPosition = RPCameraPosition.front
    } else {
        sharedRecorder.cameraPosition = RPCameraPosition.back
    }
}
Summary

In-App Screen Capture
iOS Screen Record and Broadcast
Broadcast Pairing
Fast Camera Switching
More Information

<table>
<thead>
<tr>
<th>Course</th>
<th>Location</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReplayKit 2 Lab</td>
<td>Technology Lab A</td>
<td>Fri 11:00AM–1:50PM</td>
</tr>
</tbody>
</table>