Android Developers

Managing Audio Focus

Two or more Android apps can play audio to the same output stream simultaneously. The system mixes everything together. While this is technically impressive, it can be very aggravating to a user. To avoid every music app playing at the same time, Android introduces the idea of *audio focus*. Only one app can hold audio focus at a time.

When your app needs to output audio, it should request audio focus. When it has focus, it can play sound. However, after you acquire audio focus you may not be able to keep it until you're done playing. Another app can request focus, which preempts your hold on audio focus. If that happens your app should pause playing or lower its volume to let users hear the new audio source more easily.

Audio focus is cooperative. Apps are encouraged to comply with the audio focus guidelines, but the system does not enforce the rules. If an app wants to continue to play loudly even after losing audio focus, nothing can prevent that. This is a bad experience and there's a good chance that users will uninstall an app that misbehaves in this way.

A well-behaved audio app should manage audio focus according to these general guidelines:

- Call `requestAudioFocus()` immediately before starting to play and verify that the call returns `AUDIOFOCUS_REQUEST_GRANTED` (https://developer.android.com/reference/android/media/AudioManager.html#AUDIOFOCUS_REQUEST_GRANTED). If you design your app as we describe in this guide, the call to `requestAudioFocus()` should be made in the `onPlay()` callback of your media session.
- When another app gains audio focus, stop or pause playing, or duck the volume down.
- When playback stops, abandon audio focus.

Audio focus is handled differently depending on the the version of Android that is running:

- Beginning with Android 2.2 (API level 8), apps manage audio focus by calling `requestAudioFocus()` (https://developer.android.com/reference/android/media/AudioManager.html#requestAudioFocus(android.media.AudioManager.OnAudioFocusChangeListener, int, int)) and `abandonAudioFocus()` (https://developer.android.com/reference/android/media/AudioManager.html#abandonAudioFocus(android.media.AudioManager.OnAudioFocusChangeListener)). Apps must also register an `AudioManager.OnAudioFocusChangeListener` (https://developer.android.com/reference/android/media/AudioManager.OnAudioFocusChangeListener.html) with both calls in order to receive callbacks and manage their own audio level.
- For apps that target Android 5.0 (API level 21) and later, audio apps should use `AudioAttributes` (https://developer.android.com/reference/android/media/AudioAttributes.html) to describe the type of audio your app is playing. For example, apps that play speech should specify `CONTENT_TYPE_SPEECH` (https://developer.android.com/reference/android/media/AudioAttributes.html#CONTENT_TYPE_SPEECH).
- Apps running Android 8.0 (API level 26) or greater should use the `requestAudioFocus()` (https://developer.android.com/reference/android/media/AudioManager.html#requestAudioFocus(android.media.AudioFocusRequest)) method, which takes an `AudioFocusRequest` (https://developer.android.com/reference/android/media/AudioFocusRequest.html) parameter. The `AudioFocusRequest` contains information about the audio context and capabilities of your app. The system uses this information to manage the gain and loss of audio focus automatically.

Audio focus in Android 8.0 and later

Beginning with Android 8.0 (API level 26), when you call `requestAudioFocus()` (https://developer.android.com/reference/android/media/AudioManager.html#requestAudioFocus(android.media.AudioFocusRequest)) you must supply an `AudioFocusRequest` parameter. To release audio focus, call the method `abandonAudioFocusRequest()` (https://developer.android.com/reference/android/media/AudioManager.html#abandonAudioFocusRequest(android.media.AudioFocusRequest)) which also takes an `AudioFocusRequest` as its argument. The same `AudioFocusRequest` instance should be used when requesting and abandoning focus.

To create an `AudioFocusRequest` (https://developer.android.com/reference/android/media/AudioFocusRequest.html), use an `AudioFocusRequest.Builder` (https://developer.android.com/reference/android/media/AudioFocusRequest.Builder.html). Since a focus request must always specify the type of the request, the type is included in the constructor for the builder. Use the builder's methods to set the other fields of the request.

The `FocusGain` field is required; all the other fields are optional.

Method	Notes
`setFocusGain()` (https://developer.android.com/reference/android/media/AudioFocusRequest.Builder.html#setFocusGain(int))	This field is required in every request. It takes the same values as the `durationHint` used in the pre-Android 8.0 call to `requestAudioFocus()`: `AUDIOFOCUS_GAIN`, `AUDIOFOCUS_GAIN_TRANSIENT`, `AUDIOFOCUS_GAIN_TRANSIENT_MAY_DUCK`, or `AUDIOFOCUS_GAIN_TRANSIENT_EXCLUSIVE`.

`setAudioAttributes()` (https://developer.android.com/reference/android/media/AudioFocusRequest.Builder.html#setAudioAttributes(android.media.AudioAttributes))	`AudioAttributes` (https://developer.android.com/reference/android/media/AudioAttributes.html) describes the use case for your app. The system looks at them when an app gains and loses audio focus. Attributes supersede the notion of stream type. In Android 8.0 (API level 26) and later, stream types for any operation other than volume controls are deprecated. Use the same attributes in the focus request that you use in your audio player (as shown in the example following this table). Use an `AudioAttributes.Builder` (https://developer.android.com/reference/android/media/AudioAttributes.Builder.html) to specify the attributes first, then use this method to assign the attributes to the request. If not specified, `AudioAttributes` defaults to `AudioAttributes.USAGE_MEDIA`.
`setWillPauseWhenDucked()` (https://developer.android.com/reference/android/media/AudioFocusRequest.Builder.html#setWillPauseWhenDucked(boolean))	When another app requests focus with `AUDIOFOCUS_GAIN_TRANSIENT_MAY_DUCK`, the app that has focus does not usually receive an `onAudioFocusChange()` (https://developer.android.com/reference/android/media/AudioManager.OnAudioFocusChangeListener.html#onAudioFocusChange(int)) callback because the system can do the ducking by itself (#automatic-ducking). When you need to pause playback rather than duck the volume, call `setWillPauseWhenDucked(true)` and create and set an `OnAudioFocusChangeListener`, as described in automatic ducking (#automatic-ducking).
`setAcceptsDelayedFocusGain()` (https://developer.android.com/reference/android/media/AudioFocusRequest.Builder.html#setAcceptsDelayedFocusGain(boolean))	A request for audio focus can fail when the focus is locked by another app. This method enables delayed focus gain (#delayed-focus-gain): the ability to asynchronously acquire focus when it becomes available. Note that delayed focus gain only works if you also specify an `AudioManager.OnAudioFocusChangeListener` (https://developer.android.com/reference/android/media/AudioManager.OnAudioFocusChangeListener.html) in the audio request, since your app needs to receive the callback in order to know that focus was granted.
`setOnAudioFocusChangeListener()` (https://developer.android.com/reference/android/media/AudioFocusRequest.Builder.html#setOnAudioFocusChangeListener(android.media.AudioManager.OnAudioFocusChangeListener, android.os.Handler))	An `OnAudioFocusChangeListener` is only required if you also specify `willPauseWhenDucked(true)` or `setAcceptsDelayedFocusGain(true)` in the request. There are two methods for setting the listener: one with and one without a handler argument. The handler is the thread on which the listener runs. If you do not specify a handler, the handler associated with the main `Looper` (https://developer.android.com/reference/android/os/Looper.html) is used.

The following example shows how to use an `AudioFocusRequest.Builder` to build an `AudioFocusRequest` and request and abandon audio focus:

```
mAudioManager = (AudioManager) Context.getSystemService(Context.AUDIO_SERVICE);
mPlaybackAttributes = new AudioAttributes.Builder()
        .setUsage(AudioAttributes.USAGE_GAME)
        .setContentType(AudioAttributes.CONTENT_TYPE_MUSIC)
        .build();
mFocusRequest = new AudioFocusRequest.Builder(AudioManager.AUDIOFOCUS_GAIN)
        .setAudioAttributes(mPlaybackAttributes)
        .setAcceptsDelayedFocusGain(true)
        .setOnAudioFocusChangeListener(mMyFocusListener, mMyHandler)
        .build();
mMediaPlayer = new MediaPlayer();
final Object mFocusLock = new Object();

boolean mPlaybackDelayed = false;
boolean mPlaybackNowAuthorized = false;

// ...
int res = mAudioManager.requestAudioFocus(mFocusRequest);
synchronized(mFocusLock) {
    if (res == AudioManager.AUDIOFOCUS_REQUEST_FAILED) {
        mPlaybackNowAuthorized = false;
    } else if (res == AudioManager.AUDIOFOCUS_REQUEST_GRANTED) {
        mPlaybackNowAuthorized = true;
        playbackNow();
    } else if (res == AudioManager.AUDIOFOCUS_REQUEST_DELAYED) {
       mPlaybackDelayed = true;
       mPlaybackNowAuthorized = false;
    }
}

// ...
@Override
public void onAudioFocusChange(int focusChange) {
    switch (focusChange) {
        case AudioManager.AUDIOFOCUS_GAIN:
            if (mPlaybackDelayed || mResumeOnFocusGain) {
                synchronized(mFocusLock) {
                    mPlaybackDelayed = false;
                    mResumeOnFocusGain = false;
                }
                playbackNow();
            }
            break;
        case AudioManager.AUDIOFOCUS_LOSS:
            synchronized(mFocusLock) {
                mResumeOnFocusGain = false;
                mPlaybackDelayed = false;
            }
            pausePlayback();
            break;
        case AudioManager.AUDIOFOCUS_LOSS_TRANSIENT:
            synchronized(mFocusLock) {
                mResumeOnFocusGain = true;
                mPlaybackDelayed = false;
            }
            pausePlayback();
            break;
        case AudioManager.AUDIOFOCUS_LOSS_TRANSIENT_CAN_DUCK:
            // ... pausing or ducking depends on your app
            break;
        }
    }
}
```

Automatic ducking

In Android 8.0 (API level 26), when another app requests focus with `AUDIOFOCUS_GAIN_TRANSIENT_MAY_DUCK` the system can duck and restore the volume without invoking the app's `onAudioFocusChange()` callback.

While automatic ducking is acceptable behavior for music and video playback apps, it isn't useful when playing spoken content, such as in an audio book app. In this case, the app should pause instead.

If you want your app to pause when asked to duck rather than decrease its volume, create an `OnAudioFocusChangeListener` with an `onAudioFocusChange()` callback method that implements the desired pause/resume behavior. Call `setOnAudioFocusChangeListener()` (https://developer.android.com/reference/android/media/AudioFocusRequest.Builder.html#setOnAudioFocusChangeListener(android.media.AudioManager.OnAudioFocusChangeListener, android.os.Handler)) to register the listener, and call `setWillPauseWhenDucked(true)` (https://developer.android.com/reference/android/media/AudioFocusRequest.Builder.html#setWillPauseWhenDucked(boolean)) to tell the system to use your callback rather than perform automatic ducking.

Delayed focus gain

Sometimes the system cannot grant a request for audio focus because the focus is "locked" by another app, such as during a phone call. In this case, `requestAudioFocus()` returns `AUDIOFOCUS_REQUEST_FAILED`. When this happens, your app should not proceed with audio playback because it did not gain focus.

The method, `setAcceptsDelayedFocusGain(true)` (https://developer.android.com/reference/android/media/AudioFocusRequest.Builder.html#setAcceptsDelayedFocusGain(boolean)), that lets your app handle a request for focus asynchronously. With this flag set, a request made when the focus is locked returns `AUDIOFOCUS_REQUEST_DELAYED`. When the condition that locked the audio focus no longer exists, such as when a phone call ends, the system grants the pending focus request and calls `onAudioFocusChange()` to notify your app.

In order to handle the delayed gain of focus, you must create an `OnAudioFocusChangeListener` with an `onAudioFocusChange()` callback method that implements the desired behavior and register the listener by calling `setOnAudioFocusChangeListener()` (https://developer.android.com/reference/android/media/AudioFocusRequest.Builder.html#setOnAudioFocusChangeListener(android.media.AudioManager.OnAudioFocusChangeListener, android.os.Handler)).

Audio focus pre-Android 8.0

When you call `requestAudioFocus()` (https://developer.android.com/reference/android/media/AudioManager.html#requestAudioFocus(android.media.AudioManager.OnAudioFocusChangeListener, int, int)) you must specify a duration hint, which may be honored by another app that is currently holding focus and playing:

- Request permanent audio focus (`AUDIOFOCUS_GAIN`) when you plan to play audio for the foreseeable future (for example, when playing music) and you expect the previous holder of audio focus to stop playing.
- Request transient focus (`AUDIOFOCUS_GAIN_TRANSIENT`) when you expect to play audio for only a short time and you expect the previous holder to pause playing.
- Request transient focus with *ducking* (`AUDIOFOCUS_GAIN_TRANSIENT_MAY_DUCK`) to indicate that you expect to play audio for only a short time and that it's OK for the previous focus owner to keep playing if it "ducks" (lowers) its audio output. Both audio ouputs are mixed into the audio stream. Ducking is particularly suitable for apps that use the audio stream intermittently, such as for audible driving directions.

The `requestAudioFocus()` method also requires an `AudioManager.OnAudioFocusChangeListener` (https://developer.android.com/reference/android/media/AudioManager.OnAudioFocusChangeListener.html). This listener should be created in the same activity or service that owns your media session. It implements the callback `onAudioFocusChange()` that your app receives when some other app acquires or abandons audio focus.

The following snippet requests permanent audio focus on the stream `STREAM_MUSIC` and registers an `OnAudioFocusChangeListener` to handle subsequent changes in audio focus. (The change listener is discussed in Responding to an audio focus change (#audio-focus-change).)

```
AudioManager am = (AudioManager) mContext.getSystemService(Context.AUDIO_SERVICE);
AudioManager.OnAudioFocusChangeListener afChangeListener;

...
// Request audio focus for playback
int result = am.requestAudioFocus(afChangeListener,
                             // Use the music stream.
                             AudioManager.STREAM_MUSIC,
                             // Request permanent focus.
                             AudioManager.AUDIOFOCUS_GAIN);

if (result == AudioManager.AUDIOFOCUS_REQUEST_GRANTED) {
    // Start playback
}
```

When you finish playback, call `abandonAudioFocus()` (https://developer.android.com/reference/android/media/AudioManager.html#abandonAudioFocus(android.media.AudioManager.OnAudioFocusChangeListener)).

```
// Abandon audio focus when playback complete
am.abandonAudioFocus(afChangeListener);
```

This notifies the system that you no longer require focus and unregisters the associated `OnAudioFocusChangeListener`. If you requested transient focus, this will notify an app that paused or ducked that it may continue playing or restore its volume.

Responding to an audio focus change

When an app acquires audio focus, it must be able to release it when another app requests audio focus for itself. When this happens, your app receives a call to the `onAudioFocusChange()` (https://developer.android.com/reference/android/media/AudioManager.OnAudioFocusChangeListener.html#onAudioFocusChange(int)) method in the `AudioFocusChangeListener` that you specified when the app called `requestAudioFocus()`.

The `focusChange` parameter passed to `onAudioFocusChange()` indicates the kind of change that's happening. It corresponds to the duration hint used by the app that's aquiring focus. Your app should respond appropriately.

Transient loss of focus

> If the focus change is transient (`AUDIOFOCUS_LOSS_TRANSIENT_CAN_DUCK` or `AUDIOFOCUS_LOSS_TRANSIENT`), your app should duck (if you are not relying on automatic ducking (#automatic-ducking)) or pause playing but otherwise maintain the same state.
>
> During a transient loss of audio focus, you should continue to monitor changes in audio focus and be prepared to resume normal playback when you regain the focus. When the blocking app abandons focus, you receive a callback (`AUDIOFOCUS_GAIN`). At this point, you can restore the volume to normal level or restart playback.

Permanent loss of focus

> If the audio focus loss is permanent (`AUDIOFOCUS_LOSS`), another app is playing audio. Your app should pause playback immediately, as it won't ever receive an `AUDIOFOCUS_GAIN` callback. To restart playback, the user must take an explicit action, like pressing the play transport control in a notification or app UI.

The following code snippet demonstrates how to implement the `OnAudioFocusChangeListener` and its `onAudioFocusChange()` callback. Notice the use of a `Handler` to delay the stop callback on a permanent loss of audio focus.

```
private Handler mHandler = new Handler();
AudioManager.OnAudioFocusChangeListener afChangeListener =
  new AudioManager.OnAudioFocusChangeListener() {
    public void onAudioFocusChange(int focusChange) {
      if (focusChange == AudioManager.AUDIOFOCUS_LOSS) {
        // Permanent loss of audio focus
        // Pause playback immediately
        mediaController.getTransportControls().pause();
        // Wait 30 seconds before stopping playback
        mHandler.postDelayed(mDelayedStopRunnable,
          TimeUnit.SECONDS.toMillis(30));
      }
      else if (focusChange == AUDIOFOCUS_LOSS_TRANSIENT) {
        // Pause playback
      } else if (focusChange == AUDIOFOCUS_LOSS_TRANSIENT_CAN_DUCK) {
        // Lower the volume, keep playing
      } else if (focusChange == AudioManager.AUDIOFOCUS_GAIN) {
        // Your app has been granted audio focus again
        // Raise volume to normal, restart playback if necessary
      }
    }
  };
```

The handler uses a `Runnable` that looks like this:

```
private Runnable mDelayedStopRunnable = new Runnable() {
    @Override
    public void run() {
        mediaController.getTransportControls().stop();
    }
};
```

To ensure the delayed stop does not kick in if the user restarts playback, call `mHandler.removeCallbacks(mDelayedStopRunnable)` in response to any state changes. For example, call `removeCallbacks()` in your Callback's `onPlay()`, `onSkipToNext()`, etc. You should also call this method in your service's `onDestroy()` callback when cleaning up the resources used by your service.

Follow @AndroidDev on Twitter

Follow Android Developers on Google+

Check out Android Developers on YouTube