MediaCodec

developer.android.com/reference/android/media/MediaCodec.html

```
public final class MediaCodec
extends Object
```

java.lang.Object

↳ android.media.MediaCodec

MediaCodec class can be used to access low-level media codecs, i.e. encoder/decoder components. It is part of the Android low-level multimedia support infrastructure (normally used together with `MediaExtractor`, `MediaSync`, `MediaMuxer`, `MediaCrypto`, `MediaDrm`, `Image`, `Surface`, and `AudioTrack`.)

In broad terms, a codec processes input data to generate output data. It processes data asynchronously and uses a set of input and output buffers. At a simplistic level, you request (or receive) an empty input buffer, fill it up with data and send it to the codec for processing. The codec uses up the data and transforms it into one of its empty output buffers. Finally, you request (or receive) a filled output buffer, consume its contents and release it back to the codec.

Data Types

Codecs operate on three kinds of data: compressed data, raw audio data and raw video data. All three kinds of data can be processed using `ByteBuffers`, but you should use a `Surface` for raw video data to improve codec performance. Surface uses native video buffers without mapping or copying them to ByteBuffers; thus, it is much more efficient. You normally cannot access the raw video data when using a Surface, but you can use the `ImageReader` class to access unsecured decoded (raw) video frames. This may still be more efficient than using ByteBuffers, as some native buffers may be mapped into direct ByteBuffers. When using ByteBuffer mode, you can access raw video frames using the `Image` class and `getInput`/`OutputImage(int)`.

Compressed Buffers

Input buffers (for decoders) and output buffers (for encoders) contain compressed data according to the format's type. For video types this is a single compressed video frame. For audio data this is normally a single access unit (an encoded audio segment typically containing a few milliseconds of audio as dictated by the format type), but this requirement is slightly relaxed in that a buffer may contain multiple encoded access units of audio. In either case, buffers do not start or end on arbitrary byte boundaries, but rather on frame/access unit boundaries.

Raw Audio Buffers

Raw audio buffers contain entire frames of PCM audio data, which is one sample for each channel in channel order. Each sample is a 16-bit signed integer in native byte order.

```
short[] getSamplesForChannel(MediaCodec codec, int bufferId, int channelIx) {
  ByteBuffer outputBuffer = codec.getOutputBuffer(bufferId);
  MediaFormat format = codec.getOutputFormat(bufferId);
  ShortBuffer samples = outputBuffer.order(ByteOrder.nativeOrder()).asShortBuffer();
  int numChannels = formet.getInteger(MediaFormat.KEY_CHANNEL_COUNT);
  if (channelIx < 0 || channelIx >= numChannels) {
    return null;
  }
  short[] res = new short[samples.remaining() / numChannels];
  for (int i = 0; i < res.length; ++i) {
    res[i] = samples.get(i * numChannels + channelIx);
  }
  return res;
}
```

Raw Video Buffers

In ByteBuffer mode video buffers are laid out according to their color format. You can get the supported color formats as an array from `getCodecInfo().getCapabilitiesForType(…).colorFormats`. Video codecs may support three kinds of color formats:

All video codecs support flexible YUV 4:2:0 buffers since `LOLLIPOP_MR1`.

Accessing Raw Video ByteBuffers on Older Devices

Prior to `LOLLIPOP` and `Image` support, you need to use the `KEY_STRIDE` and `KEY_SLICE_HEIGHT` output format values to understand the layout of the raw output buffers.

Note that on some devices the slice-height is advertised as 0. This could mean either that the slice-height is the same as the frame height, or that the slice-height is the frame height aligned to some value (usually a power of 2). Unfortunately, there is no standard and simple way to tell the actual slice height in this case. Furthermore, the vertical stride of the `U` plane in planar formats is also not specified or defined, though usually it is half of the slice height.

The `KEY_WIDTH` and `KEY_HEIGHT` keys specify the size of the video frames; however, for most encondings the video (picture) only occupies a portion of the video frame. This is represented by the 'crop rectangle'.

You need to use the following keys to get the crop rectangle of raw output images from the output format. If these keys are not present, the video occupies the entire video frame.The crop rectangle is understood in the context of the output frame *before* applying any rotation.

Format Key	Type	Description
`"crop-left"`	Integer	The left-coordinate (x) of the crop rectangle
`"crop-top"`	Integer	The top-coordinate (y) of the crop rectangle

Format Key	Type	Description
`"crop-right"`	Integer	The right-coordinate (x) **MINUS 1** of the crop rectangle
`"crop-bottom"`	Integer	The bottom-coordinate (y) **MINUS 1** of the crop rectangle

The right and bottom coordinates can be understood as the coordinates of the right-most valid column/bottom-most valid row of the cropped output image.

The size of the video frame (before rotation) can be calculated as such:

```
MediaFormat format = decoder.getOutputFormat(…);
int width = format.getInteger(MediaFormat.KEY_WIDTH);
if (format.containsKey("crop-left") && format.containsKey("crop-right")) {
    width = format.getInteger("crop-right") + 1 - format.getInteger("crop-left");
}
int height = format.getInteger(MediaFormat.KEY_HEIGHT);
if (format.containsKey("crop-top") && format.containsKey("crop-bottom")) {
    height = format.getInteger("crop-bottom") + 1 - format.getInteger("crop-top");
}
```

Also note that the meaning of `BufferInfo.offset` was not consistent across devices. On some devices the offset pointed to the top-left pixel of the crop rectangle, while on most devices it pointed to the top-left pixel of the entire frame.

States

During its life a codec conceptually exists in one of three states: Stopped, Executing or Released. The Stopped collective state is actually the conglomeration of three states: Uninitialized, Configured and Error, whereas the Executing state conceptually progresses through three sub-states: Flushed, Running and End-of-Stream.

When you create a codec using one of the factory methods, the codec is in the Uninitialized state. First, you need to configure it via `configure(…)`, which brings it to the Configured state, then call `start()` to move it to the Executing state. In this state you can process data through the buffer queue manipulation described above.

The Executing state has three sub-states: Flushed, Running and End-of-Stream. Immediately after `start()` the codec is in the Flushed sub-state, where it holds all the buffers. As soon as the first input buffer is dequeued, the codec moves to the Running sub-state, where it spends most of its life. When you queue an input buffer with the end-of-stream marker, the codec transitions to the End-of-Stream sub-state. In this state the codec no longer accepts further input buffers, but still generates output buffers until the end-of-stream is reached on the output. You can move back to the Flushed sub-state at any time while in the Executing state using `flush()`.

Call `stop()` to return the codec to the Uninitialized state, whereupon it may be configured again. When you are done using a codec, you must release it by calling `release()`.

On rare occasions the codec may encounter an error and move to the Error state. This is communicated using an invalid return value from a queuing operation, or sometimes via an exception. Call `reset()` to make the codec usable again. You can call it from any state to move the codec back to the Uninitialized state. Otherwise, call `release()` to move to the terminal Released state.

Creation

Use `MediaCodecList` to create a MediaCodec for a specific `MediaFormat`. When decoding a file or a stream, you can get the desired format from `MediaExtractor.getTrackFormat`. Inject any specific features that you want to add using `MediaFormat.setFeatureEnabled`, then call `MediaCodecList.findDecoderForFormat` to get the name of a codec that can handle that specific media format. Finally, create the codec using `createByCodecName(String)`.

Note: On `LOLLIPOP`, the format to `MediaCodecList.findDecoder/EncoderForFormat` must not contain a frame rate. Use `format.setString(MediaFormat.KEY_FRAME_RATE, null)` to clear any existing frame rate setting in the format.

You can also create the preferred codec for a specific MIME type using `createDecoder/EncoderByType(String)`. This, however, cannot be used to inject features, and may create a codec that cannot handle the specific desired media format.

Creating secure decoders

On versions `KITKAT_WATCH` and earlier, secure codecs might not be listed in `MediaCodecList`, but may still be available on the system. Secure codecs that exist can be instantiated by name only, by appending `".secure"` to the name of a regular codec (the name of all secure codecs must end in `".secure"`.) `createByCodecName(String)` will throw an `IOException` if the codec is not present on the system.

From `LOLLIPOP` onwards, you should use the `FEATURE_SecurePlayback` feature in the media format to create a secure decoder.

Initialization

After creating the codec, you can set a callback using `setCallback` if you want to process data asynchronously. Then, configure the codec using the specific media format. This is when you can specify the output `Surface` for video producers – codecs that generate raw video data (e.g. video decoders). This is also when you can set the decryption parameters for secure codecs (see `MediaCrypto`). Finally, since some codecs can operate in multiple modes, you must specify whether you want it to work as a decoder or an encoder.

Since `LOLLIPOP`, you can query the resulting input and output format in the Configured state. You can use this to verify the resulting configuration, e.g. color formats, before starting the codec.

If you want to process raw input video buffers natively with a video consumer – a codec that processes raw video input, such as a video encoder – create a destination Surface for your input data using `createInputSurface()` after configuration. Alternately, set up the codec to use a previously created persistent input surface by calling `setInputSurface(Surface)`.

Codec-specific Data

Some formats, notably AAC audio and MPEG4, H.264 and H.265 video formats require the actual data to be prefixed by a number of buffers containing setup data, or codec specific data. When processing such compressed formats, this data must be submitted to the codec after `start()` and before any frame data. Such data must be marked using the flag `BUFFER_FLAG_CODEC_CONFIG` in a call to `queueInputBuffer`.

Codec-specific data can also be included in the format passed to `configure` in ByteBuffer entries with keys "csd-0", "csd-1", etc. These keys are always included in the track `MediaFormat` obtained from the `MediaExtractor`. Codec-specific data in the format is automatically submitted to the codec upon `start()`; you **MUST NOT** submit this data explicitly. If the format did not contain codec specific data, you can choose to submit it using the specified number of buffers in the correct order, according to the format requirements. In case of H.264 AVC, you can also concatenate all codec-specific data and submit it as a single codec-config buffer.

Android uses the following codec-specific data buffers. These are also required to be set in the track format for proper `MediaMuxer` track configuration. Each parameter set and the codec-specific-data sections marked with (*) must start with a start code of `"\x00\x00\x00\x01"`.

Format	CSD buffer #0	CSD buffer #1	CSD buffer #2
AAC	Decoder-specific information from ESDS*	Not Used	Not Used
VORBIS	Identification header	Setup header	Not Used
OPUS	Identification header	Pre-skip in nanosecs (unsigned 64-bit native-order integer.) This overrides the pre-skip value in the identification header.	Seek Pre-roll in nanosecs (unsigned 64-bit native-order integer.)
MPEG-4	Decoder-specific information from ESDS*	Not Used	Not Used
H.264 AVC	SPS (Sequence Parameter Sets*)	PPS (Picture Parameter Sets*)	Not Used
H.265 HEVC	VPS (Video Parameter Sets*) + SPS (Sequence Parameter Sets*) + PPS (Picture Parameter Sets*)	Not Used	Not Used
VP9	VP9 CodecPrivate Data (optional)	Not Used	Not Used

Note: care must be taken if the codec is flushed immediately or shortly after start, before any output buffer or output format change has been returned, as the codec specific data may be lost during the flush. You must resubmit the data using buffers marked with `BUFFER_FLAG_CODEC_CONFIG` after such flush to ensure proper codec operation.

Encoders (or codecs that generate compressed data) will create and return the codec specific data before any valid output buffer in output buffers marked with the codec-config flag. Buffers containing codec-specific-data have no meaningful timestamps.

Data Processing

Each codec maintains a set of input and output buffers that are referred to by a buffer-ID in API calls. After a successful call to `start()` the client "owns" neither input nor output buffers. In synchronous mode, call `dequeueInput`/`OutputBuffer(…)` to obtain (get ownership of) an input or output buffer from the codec. In asynchronous mode, you will automatically receive available buffers via the `MediaCodec.Callback.onInput`/`OutputBufferAvailable(…)` callbacks.

Upon obtaining an input buffer, fill it with data and submit it to the codec using `queueInputBuffer` – or `queueSecureInputBuffer` if using decryption. Do not submit multiple input buffers with the same timestamp (unless it is codec-specific data marked as such).

The codec in turn will return a read-only output buffer via the `onOutputBufferAvailable` callback in asynchronous mode, or in response to a `dequeuOutputBuffer` call in synchronous mode. After the output buffer has been processed, call one of the `releaseOutputBuffer` methods to return the buffer to the codec.

While you are not required to resubmit/release buffers immediately to the codec, holding onto input and/or output buffers may stall the codec, and this behavior is device dependent. **Specifically, it is possible that a codec may hold off on generating output buffers until *all* outstanding buffers have been released/resubmitted.** Therefore, try to hold onto to available buffers as little as possible.

Depending on the API version, you can process data in three ways:

Processing Mode	API version <= 20 Jelly Bean/KitKat	API version >= 21 Lollipop and later
Synchronous API using buffer arrays	Supported	Deprecated
Synchronous API using buffers	Not Available	Supported
Asynchronous API using buffers	Not Available	Supported

Asynchronous Processing using Buffers

Since `LOLLIPOP`, the preferred method is to process data asynchronously by setting a callback before calling `configure`. Asynchronous mode changes the state transitions slightly, because you must call `start()` after `flush()` to transition the codec to the Running sub-state and start receiving input buffers. Similarly, upon an initial call to `start` the codec will move directly to the Running sub-state and start passing available input buffers via the callback.

MediaCodec is typically used like this in asynchronous mode:

```
MediaCodec codec = MediaCodec.createByCodecName(name);
MediaFormat mOutputFormat; // member variable
codec.setCallback(new MediaCodec.Callback() {
  @Override
  void onInputBufferAvailable(MediaCodec mc, int inputBufferId) {
    ByteBuffer inputBuffer = codec.getInputBuffer(inputBufferId);
    // fill inputBuffer with valid data
    …
    codec.queueInputBuffer(inputBufferId, …);
  }

  @Override
```

```
  void onOutputBufferAvailable(MediaCodec mc, int outputBufferId, …) {
    ByteBuffer outputBuffer = codec.getOutputBuffer(outputBufferId);
    MediaFormat bufferFormat = codec.getOutputFormat(outputBufferId); // option A
    // bufferFormat is equivalent to mOutputFormat
    // outputBuffer is ready to be processed or rendered.
    …
    codec.releaseOutputBuffer(outputBufferId, …);
  }

  @Override
  void onOutputFormatChanged(MediaCodec mc, MediaFormat format) {
    // Subsequent data will conform to new format.
    // Can ignore if using getOutputFormat(outputBufferId)
    mOutputFormat = format; // option B
  }

  @Override
  void onError(…) {
    …
  }
});
codec.configure(format, …);
mOutputFormat = codec.getOutputFormat(); // option B
codec.start();
// wait for processing to complete
codec.stop();
codec.release();
```

Synchronous Processing using Buffers

Since `LOLLIPOP`, you should retrieve input and output buffers using `getInput/OutputBuffer(int)` and/or `getInput/OutputImage(int)` even when using the codec in synchronous mode. This allows certain optimizations by the framework, e.g. when processing dynamic content. This optimization is disabled if you call `getInput/OutputBuffers()`.

Note: do not mix the methods of using buffers and buffer arrays at the same time. Specifically, only call `getInput/OutputBuffers` directly after `start()` or after having dequeued an output buffer ID with the value of `INFO_OUTPUT_FORMAT_CHANGED`.

MediaCodec is typically used like this in synchronous mode:

```
MediaCodec codec = MediaCodec.createByCodecName(name);
codec.configure(format, …);
MediaFormat outputFormat = codec.getOutputFormat(); // option B
codec.start();
for (;;) {
  int inputBufferId = codec.dequeueInputBuffer(timeoutUs);
  if (inputBufferId >= 0) {
    ByteBuffer inputBuffer = codec.getInputBuffer(…);
    // fill inputBuffer with valid data
    …
    codec.queueInputBuffer(inputBufferId, …);
  }
  int outputBufferId = codec.dequeueOutputBuffer(…);
  if (outputBufferId >= 0) {
    ByteBuffer outputBuffer = codec.getOutputBuffer(outputBufferId);
    MediaFormat bufferFormat = codec.getOutputFormat(outputBufferId); // option A
    // bufferFormat is identical to outputFormat
    // outputBuffer is ready to be processed or rendered.
    …
    codec.releaseOutputBuffer(outputBufferId, …);
  } else if (outputBufferId == MediaCodec.INFO_OUTPUT_FORMAT_CHANGED) {
    // Subsequent data will conform to new format.
    // Can ignore if using getOutputFormat(outputBufferId)
    outputFormat = codec.getOutputFormat(); // option B
  }
}
codec.stop();
codec.release();
```

Synchronous Processing using Buffer Arrays (deprecated)

In versions `KITKAT_WATCH` and before, the set of input and output buffers are represented by the `ByteBuffer[]` arrays. After a successful call to `start()`, retrieve the buffer arrays using `getInput/OutputBuffers()`. Use the buffer ID-s as indices into these arrays (when non-negative), as demonstrated in the sample below. Note that there is no inherent correlation between the size of the arrays and the number of input and output buffers used by the system, although the array size provides an upper bound.

```
MediaCodec codec = MediaCodec.createByCodecName(name);
codec.configure(format, …);
codec.start();
ByteBuffer[] inputBuffers = codec.getInputBuffers();
ByteBuffer[] outputBuffers = codec.getOutputBuffers();
for (;;) {
  int inputBufferId = codec.dequeueInputBuffer(…);
  if (inputBufferId >= 0) {
    // fill inputBuffers[inputBufferId] with valid data
    …
    codec.queueInputBuffer(inputBufferId, …);
  }
```

```
  int outputBufferId = codec.dequeueOutputBuffer(…);
  if (outputBufferId >= 0) {
    // outputBuffers[outputBufferId] is ready to be processed or rendered.
    …
    codec.releaseOutputBuffer(outputBufferId, …);
  } else if (outputBufferId == MediaCodec.INFO_OUTPUT_BUFFERS_CHANGED) {
    outputBuffers = codec.getOutputBuffers();
  } else if (outputBufferId == MediaCodec.INFO_OUTPUT_FORMAT_CHANGED) {
    // Subsequent data will conform to new format.
    MediaFormat format = codec.getOutputFormat();
  }
}
codec.stop();
codec.release();
```

End-of-stream Handling

When you reach the end of the input data, you must signal it to the codec by specifying the `BUFFER_FLAG_END_OF_STREAM` flag in the call to `queueInputBuffer`. You can do this on the last valid input buffer, or by submitting an additional empty input buffer with the end-of-stream flag set. If using an empty buffer, the timestamp will be ignored.

The codec will continue to return output buffers until it eventually signals the end of the output stream by specifying the same end-of-stream flag in the `MediaCodec.BufferInfo` set in `dequeueOutputBuffer` or returned via `onOutputBufferAvailable`. This can be set on the last valid output buffer, or on an empty buffer after the last valid output buffer. The timestamp of such empty buffer should be ignored.

Do not submit additional input buffers after signaling the end of the input stream, unless the codec has been flushed, or stopped and restarted.

Using an Output Surface

The data processing is nearly identical to the ByteBuffer mode when using an output `Surface`; however, the output buffers will not be accessible, and are represented as `null` values. E.g. `getOutputBuffer`/`Image(int)` will return `null` and `getOutputBuffers()` will return an array containing only `null`-s.

When using an output Surface, you can select whether or not to render each output buffer on the surface. You have three choices:

Since `M`, the default timestamp is the presentation timestamp of the buffer (converted to nanoseconds). It was not defined prior to that.

Also since `M`, you can change the output Surface dynamically using `setOutputSurface`.

Transformations When Rendering onto Surface

If the codec is configured into Surface mode, any crop rectangle, rotation and video scaling mode will be automatically applied with one exception:

Prior to the `M` release, software decoders may not have applied the rotation when being rendered onto a Surface. Unfortunately, there is no standard and simple way to identify software decoders, or if they apply the rotation other than by trying it out.

There are also some caveats.

Note that the pixel aspect ratio is not considered when displaying the output onto the Surface. This means that if you are using `VIDEO_SCALING_MODE_SCALE_TO_FIT` mode, you must position the output Surface so that it has the proper final display aspect ratio. Conversely, you can only use `VIDEO_SCALING_MODE_SCALE_TO_FIT_WITH_CROPPING` mode for content with square pixels (pixel aspect ratio or 1:1).

Note also that as of `N` release, `VIDEO_SCALING_MODE_SCALE_TO_FIT_WITH_CROPPING` mode may not work correctly for videos rotated by 90 or 270 degrees.

When setting the video scaling mode, note that it must be reset after each time the output buffers change. Since the `INFO_OUTPUT_BUFFERS_CHANGED` event is deprecated, you can do this after each time the output format changes.

Using an Input Surface

When using an input Surface, there are no accessible input buffers, as buffers are automatically passed from the input surface to the codec. Calling `dequeueInputBuffer` will throw an `IllegalStateException`, and `getInputBuffers()` returns a bogus `ByteBuffer[]` array that **MUST NOT** be written into.

Call `signalEndOfInputStream()` to signal end-of-stream. The input surface will stop submitting data to the codec immediately after this call.

Seeking & Adaptive Playback Support

Video decoders (and in general codecs that consume compressed video data) behave differently regarding seek and format change whether or not they support and are configured for adaptive playback. You can check if a decoder supports adaptive playback via `CodecCapabilities.isFeatureSupported(String)`. Adaptive playback support for video decoders is only activated if you configure the codec to decode onto a `Surface`.

Stream Boundary and Key Frames

It is important that the input data after `start()` or `flush()` starts at a suitable stream boundary: the first frame must a key frame. A *key frame* can be decoded completely on its own (for most codecs this means an I-frame), and no frames that are to be displayed after a key frame refer to frames before the key frame.

The following table summarizes suitable key frames for various video formats.

Format	Suitable key frame
VP9/VP8	a suitable intraframe where no subsequent frames refer to frames prior to this frame. *(There is no specific name for such key frame.)*
H.265 HEVC	IDR or CRA
H.264 AVC	IDR

Format	Suitable key frame
MPEG-4 H.263 MPEG-2	a suitable I-frame where no subsequent frames refer to frames prior to this frame. *(There is no specific name for such key frame.)*

For decoders that do not support adaptive playback (including when not decoding onto a Surface)

In order to start decoding data that is not adjacent to previously submitted data (i.e. after a seek) you **MUST** flush the decoder. Since all output buffers are immediately revoked at the point of the flush, you may want to first signal then wait for the end-of-stream before you call `flush`. It is important that the input data after a flush starts at a suitable stream boundary/key frame.

Note: the format of the data submitted after a flush must not change; `flush()` does not support format discontinuities; for that, a full `stop()` - `configure(…)` - `start()` cycle is necessary.

Also note: if you flush the codec too soon after `start()` – generally, before the first output buffer or output format change is received – you will need to resubmit the codec-specific-data to the codec. See the codec-specific-data section for more info.

For decoders that support and are configured for adaptive playback

In order to start decoding data that is not adjacent to previously submitted data (i.e. after a seek) it is *not necessary* to flush the decoder; however, input data after the discontinuity must start at a suitable stream boundary/key frame.

For some video formats - namely H.264, H.265, VP8 and VP9 - it is also possible to change the picture size or configuration mid-stream. To do this you must package the entire new codec-specific configuration data together with the key frame into a single buffer (including any start codes), and submit it as a **regular** input buffer.

You will receive an `INFO_OUTPUT_FORMAT_CHANGED` return value from `dequeueOutputBuffer` or a `onOutputFormatChanged` callback just after the picture-size change takes place and before any frames with the new size have been returned.

Note: just as the case for codec-specific data, be careful when calling `flush()` shortly after you have changed the picture size. If you have not received confirmation of the picture size change, you will need to repeat the request for the new picture size.

Error handling

The factory methods `createByCodecName` and `createDecoder`/`EncoderByType` throw `IOException` on failure which you must catch or declare to pass up. MediaCodec methods throw `IllegalStateException` when the method is called from a codec state that does not allow it; this is typically due to incorrect application API usage. Methods involving secure buffers may throw `MediaCodec.CryptoException`, which has further error information obtainable from `getErrorCode()`.

Internal codec errors result in a `MediaCodec.CodecException`, which may be due to media content corruption, hardware failure, resource exhaustion, and so forth, even when the application is correctly using the API. The recommended action when receiving a `CodecException` can be determined by calling `isRecoverable()` and `isTransient()`:

- **recoverable errors:** If `isRecoverable()` returns true, then call `stop()`, `configure(…)`, and `start()` to recover.
- **transient errors:** If `isTransient()` returns true, then resources are temporarily unavailable and the method may be retried at a later time.
- **fatal errors:** If both `isRecoverable()` and `isTransient()` return false, then the `CodecException` is fatal and the codec must be reset or released.

Both `isRecoverable()` and `isTransient()` do not return true at the same time.

Valid API Calls and API History

This sections summarizes the valid API calls in each state and the API history of the MediaCodec class. For API version numbers, see `Build.VERSION_CODES`.

Symbol	Meaning
•	Supported
⁕	Semantics changed
○	Experimental support
[ ]	Deprecated
⍉	Restricted to surface input mode
⍈	Restricted to surface output mode
▧	Restricted to ByteBuffer input mode
↩	Restricted to synchronous mode
⇄	Restricted to asynchronous mode
( )	Can be called, but shouldn't

Uninitialized	Configured	Flushed	Running	End of Stream	Error	Released	Method	SDK Version							
State								16	17	18	19	20	21	22	23
							createByCodecName	●	●	●	●	●	●	●	●
							createDecoderByType	●	●	●	●	●	●	●	●
							createEncoderByType	●	●	●	●	●	●	●	●
							createPersistentInputSurface								●
16+	-	-	-	-	-	-	configure	●	●	●	●	●	*	●	●
-	18+	-	-	-	-	-	createInputSurface			⃠	⃠	⃠	⃠	⃠	⃠
-	-	16+	16+	(16+)	-	-	dequeueInputBuffer	●	●	▧	▧	▧	*▧↩	▧↩	▧↩
-	-	16+	16+	16+	-	-	dequeueOutputBuffer	●	●	●	●	●	*↩	↩	↩
-	-	16+	16+	16+	-	-	flush	●	●	●	●	●	●	●	●
18+	18+	18+	18+	18+	18+	-	getCodecInfo			●	●	●	●	●	●
-	-	(21+)	21+	(21+)	-	-	getInputBuffer						●	●	●
-	-	16+	(16+)	(16+)	-	-	getInputBuffers	●	●	●	●	●	[*↩]	[↩]	[↩]
-	21+	(21+)	(21+)	(21+)	-	-	getInputFormat						●	●	●
-	-	(21+)	21+	(21+)	-	-	getInputImage						○	●	●
18+	18+	18+	18+	18+	18+	-	getName			●	●	●	●	●	●
-	-	(21+)	21+	21+	-	-	getOutputBuffer						●	●	●
-	-	16+	16+	16+	-	-	getOutputBuffers	●	●	●	●	●	[*↩]	[↩]	[↩]
-	21+	16+	16+	16+	-	-	getOutputFormat()	●	●	●	●	●	●	●	●
-	-	(21+)	21+	21+	-	-	getOutputFormat(int)						●	●	●
-	-	(21+)	21+	21+	-	-	getOutputImage						○	●	●
-	-	-	16+	(16+)	-	-	queueInputBuffer	●	●	●	●	●	*	●	●
-	-	-	16+	(16+)	-	-	queueSecureInputBuffer	●	●	●	●	●	*	●	●
16+	16+	16+	16+	16+	16+	16+	release	●	●	●	●	●	●	●	●
-	-	-	16+	16+	-	-	releaseOutputBuffer(int, boolean)	●	●	●	●	●	*	●	*
-	-	-	21+	21+	-	-	releaseOutputBuffer(int, long)						◈	◈	◈
21+	21+	21+	21+	21+	21+	-	reset						●	●	●
21+	-	-	-	-	-	-	setCallback						●	●	*
-	23+	-	-	-	-	-	setInputSurface								⃠
23+	23+	23+	23+	23+	(23+)	(23+)	setOnFrameRenderedListener								○ ◈
-	23+	23+	23+	23+	-	-	setOutputSurface								◈
19+	19+	19+	19+	19+	(19+)	-	setParameters				●	●	●	●	●
-	(16+)	(16+)	16+	(16+)	(16+)	-	setVideoScalingMode	◈	◈	◈	◈	◈	◈	◈	◈
-	-	18+	18+	-	-	-	signalEndOfInputStream			⃠	⃠	⃠	⃠	⃠	⃠
-	16+	21+(⇄)	-	-	-	-	start	●	●	●	●	●	*	●	●
-	-	16+	16+	16+	-	-	stop	●	●	●	●	●	●	●	●

Summary

Nested classes

class	MediaCodec.BufferInfo Per buffer metadata includes an offset and size specifying the range of valid data in the associated codec (output) buffer.
class	MediaCodec.Callback MediaCodec callback interface.
class	MediaCodec.CodecException Thrown when an internal codec error occurs.
class	MediaCodec.CryptoException Thrown when a crypto error occurs while queueing a secure input buffer.
class	MediaCodec.CryptoInfo Metadata describing the structure of a (at least partially) encrypted input sample.
interface	MediaCodec.OnFrameRenderedListener Listener to be called when an output frame has rendered on the output surface

Constants

int	BUFFER_FLAG_CODEC_CONFIG This indicated that the buffer marked as such contains codec initialization / codec specific data instead of media data.
int	BUFFER_FLAG_END_OF_STREAM This signals the end of stream, i.e.
int	BUFFER_FLAG_KEY_FRAME This indicates that the (encoded) buffer marked as such contains the data for a key frame.
int	BUFFER_FLAG_SYNC_FRAME *This constant was deprecated in API level 21. Use BUFFER_FLAG_KEY_FRAME instead.*
int	CONFIGURE_FLAG_ENCODE If this codec is to be used as an encoder, pass this flag.
int	CRYPTO_MODE_AES_CBC
int	CRYPTO_MODE_AES_CTR
int	CRYPTO_MODE_UNENCRYPTED
int	INFO_OUTPUT_BUFFERS_CHANGED *This constant was deprecated in API level 21. This return value can be ignored as getOutputBuffers() has been deprecated. Client should request a current buffer using on of the get-buffer or get-image methods each time one has been dequeued.*
int	INFO_OUTPUT_FORMAT_CHANGED The output format has changed, subsequent data will follow the new format.
int	INFO_TRY_AGAIN_LATER If a non-negative timeout had been specified in the call to dequeueOutputBuffer(MediaCodec.BufferInfo, long), indicates that the call timed out.
String	PARAMETER_KEY_REQUEST_SYNC_FRAME Request that the encoder produce a sync frame "soon".
String	PARAMETER_KEY_SUSPEND Temporarily suspend/resume encoding of input data.
String	PARAMETER_KEY_VIDEO_BITRATE Change a video encoder's target bitrate on the fly.

int	VIDEO_SCALING_MODE_SCALE_TO_FIT The content is scaled to the surface dimensions
int	VIDEO_SCALING_MODE_SCALE_TO_FIT_WITH_CROPPING The content is scaled, maintaining its aspect ratio, the whole surface area is used, content may be cropped.

Public methods

void	configure(MediaFormat format, Surface surface, MediaCrypto crypto, int flags) Configures a component.
static MediaCodec	createByCodecName(String name) If you know the exact name of the component you want to instantiate use this method to instantiate it.
static MediaCodec	createDecoderByType(String type) Instantiate the preferred decoder supporting input data of the given mime type.
static MediaCodec	createEncoderByType(String type) Instantiate the preferred encoder supporting output data of the given mime type.
final Surface	createInputSurface() Requests a Surface to use as the input to an encoder, in place of input buffers.
static Surface	createPersistentInputSurface() Create a persistent input surface that can be used with codecs that normally have an input surface, such as video encoders.
final int	dequeueInputBuffer(long timeoutUs) Returns the index of an input buffer to be filled with valid data or -1 if no such buffer is currently available.
final int	dequeueOutputBuffer(MediaCodec.BufferInfo info, long timeoutUs) Dequeue an output buffer, block at most "timeoutUs" microseconds.
final void	flush() Flush both input and output ports of the component.
MediaCodecInfo	getCodecInfo() Get the codec info.
ByteBuffer	getInputBuffer(int index) Returns a cleared, writable ByteBuffer object for a dequeued input buffer index to contain the input data.
ByteBuffer[]	getInputBuffers() *This method was deprecated in API level 21. Use the new getInputBuffer(int) method instead each time an input buffer is dequeued.* ***Note:*** *As of API 21, dequeued input buffers are automatically cleared. Do not use this method if using an input surface.*
final MediaFormat	getInputFormat() Call this after configure(MediaFormat, Surface, MediaCrypto, int) returns successfully to get the input format accepted by the codec.
Image	getInputImage(int index) Returns a writable Image object for a dequeued input buffer index to contain the raw input video frame.
final String	getName() Get the component name.
ByteBuffer	getOutputBuffer(int index) Returns a read-only ByteBuffer for a dequeued output buffer index.
ByteBuffer[]	getOutputBuffers() *This method was deprecated in API level 21. Use the new getOutputBuffer(int) method instead each time an output buffer is dequeued. This method is not supported if codec is configured in asynchronous mode.* ***Note:*** *As of API 21, the position and limit of output buffers that are dequeued will be set to the valid data range. Do not use this method if using an output surface.*
final MediaFormat	getOutputFormat(int index) Returns the output format for a specific output buffer.

final MediaFormat	getOutputFormat() Call this after dequeueOutputBuffer signals a format change by returning INFO_OUTPUT_FORMAT_CHANGED.
Image	getOutputImage(int index) Returns a read-only Image object for a dequeued output buffer index that contains the raw video frame.
final void	queueInputBuffer(int index, int offset, int size, long presentationTimeUs, int flags) After filling a range of the input buffer at the specified index submit it to the component.
final void	queueSecureInputBuffer(int index, int offset, MediaCodec.CryptoInfo info, long presentationTimeUs, int flags) Similar to queueInputBuffer but submits a buffer that is potentially encrypted.
final void	release() Free up resources used by the codec instance.
final void	releaseOutputBuffer(int index, boolean render) If you are done with a buffer, use this call to return the buffer to the codec or to render it on the output surface.
final void	releaseOutputBuffer(int index, long renderTimestampNs) If you are done with a buffer, use this call to update its surface timestamp and return it to the codec to render it on the output surface.
final void	reset() Returns the codec to its initial (Uninitialized) state.
void	setCallback(MediaCodec.Callback cb, Handler handler) Sets an asynchronous callback for actionable MediaCodec events.
void	setCallback(MediaCodec.Callback cb) Sets an asynchronous callback for actionable MediaCodec events on the default looper.
void	setInputSurface(Surface surface) Configures the codec (e.g.
void	setOnFrameRenderedListener(MediaCodec.OnFrameRenderedListener listener, Handler handler) Registers a callback to be invoked when an output frame is rendered on the output surface.
void	setOutputSurface(Surface surface) Dynamically sets the output surface of a codec.
final void	setParameters(Bundle params) Communicate additional parameter changes to the component instance.
final void	setVideoScalingMode(int mode) If a surface has been specified in a previous call to configure(MediaFormat, Surface, MediaCrypto, int) specifies the scaling mode to use.
final void	signalEndOfInputStream() Signals end-of-stream on input.
final void	start() After successfully configuring the component, call start.
final void	stop() Finish the decode/encode session, note that the codec instance remains active and ready to be start()ed again.

Protected methods

void	finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

Inherited methods

From class java.lang.Object

Constants

BUFFER_FLAG_CODEC_CONFIG

Added in API level 16

```
int BUFFER_FLAG_CODEC_CONFIG
```

This indicated that the buffer marked as such contains codec initialization / codec specific data instead of media data.

Constant Value: 2 (0x00000002)

BUFFER_FLAG_END_OF_STREAM

Added in API level 16

```
int BUFFER_FLAG_END_OF_STREAM
```

This signals the end of stream, i.e. no buffers will be available after this, unless of course, `flush()` follows.

Constant Value: 4 (0x00000004)

BUFFER_FLAG_KEY_FRAME

Added in API level 21

```
int BUFFER_FLAG_KEY_FRAME
```

This indicates that the (encoded) buffer marked as such contains the data for a key frame.

Constant Value: 1 (0x00000001)

BUFFER_FLAG_SYNC_FRAME

Added in API level 16

```
int BUFFER_FLAG_SYNC_FRAME
```

This constant was deprecated in API level 21.
Use `BUFFER_FLAG_KEY_FRAME` instead.

This indicates that the (encoded) buffer marked as such contains the data for a key frame.

Constant Value: 1 (0x00000001)

CONFIGURE_FLAG_ENCODE

Added in API level 16

```
int CONFIGURE_FLAG_ENCODE
```

If this codec is to be used as an encoder, pass this flag.

Constant Value: 1 (0x00000001)

CRYPTO_MODE_AES_CBC

Added in API level 24

```
int CRYPTO_MODE_AES_CBC
```

Constant Value: 2 (0x00000002)

CRYPTO_MODE_AES_CTR

Added in API level 16

```
int CRYPTO_MODE_AES_CTR
```

Constant Value: 1 (0x00000001)

CRYPTO_MODE_UNENCRYPTED

Added in API level 16

```
int CRYPTO_MODE_UNENCRYPTED
```

Constant Value: 0 (0x00000000)

INFO_OUTPUT_BUFFERS_CHANGED

Added in API level 16

```
int INFO_OUTPUT_BUFFERS_CHANGED
```

This constant was deprecated in API level 21.
This return value can be ignored as `getOutputBuffers()` has been deprecated. Client should request a current buffer using on of the get-buffer or get-image methods each time

one has been dequeued.

The output buffers have changed, the client must refer to the new set of output buffers returned by `getOutputBuffers()` from this point on.

Additionally, this event signals that the video scaling mode may have been reset to the default.

Constant Value: -3 (0xfffffffd)

INFO_OUTPUT_FORMAT_CHANGED

Added in API level 16

```
int INFO_OUTPUT_FORMAT_CHANGED
```

The output format has changed, subsequent data will follow the new format. `getOutputFormat()` returns the new format. Note, that you can also use the new `getOutputFormat(int)` method to get the format for a specific output buffer. This frees you from having to track output format changes.

Constant Value: -2 (0xfffffffe)

INFO_TRY_AGAIN_LATER

Added in API level 16

```
int INFO_TRY_AGAIN_LATER
```

If a non-negative timeout had been specified in the call to `dequeueOutputBuffer(MediaCodec.BufferInfo, long)`, indicates that the call timed out.

Constant Value: -1 (0xffffffff)

PARAMETER_KEY_REQUEST_SYNC_FRAME

Added in API level 19

```
String PARAMETER_KEY_REQUEST_SYNC_FRAME
```

Request that the encoder produce a sync frame "soon". Provide an Integer with the value 0.

Constant Value: "request-sync"

PARAMETER_KEY_SUSPEND

Added in API level 19

```
String PARAMETER_KEY_SUSPEND
```

Temporarily suspend/resume encoding of input data. While suspended input data is effectively discarded instead of being fed into the encoder. This parameter really only makes sense to use with an encoder in "surface-input" mode, as the client code has no control over the input-side of the encoder in that case. The value is an Integer object containing the value 1 to suspend or the value 0 to resume.

Constant Value: "drop-input-frames"

PARAMETER_KEY_VIDEO_BITRATE

Added in API level 19

```
String PARAMETER_KEY_VIDEO_BITRATE
```

Change a video encoder's target bitrate on the fly. The value is an Integer object containing the new bitrate in bps.

Constant Value: "video-bitrate"

VIDEO_SCALING_MODE_SCALE_TO_FIT

Added in API level 16

```
int VIDEO_SCALING_MODE_SCALE_TO_FIT
```

The content is scaled to the surface dimensions

Constant Value: 1 (0x00000001)

VIDEO_SCALING_MODE_SCALE_TO_FIT_WITH_CROPPING

Added in API level 16

```
int VIDEO_SCALING_MODE_SCALE_TO_FIT_WITH_CROPPING
```

The content is scaled, maintaining its aspect ratio, the whole surface area is used, content may be cropped.

This mode is only suitable for content with 1:1 pixel aspect ratio as you cannot configure the pixel aspect ratio for a `Surface`.

As of `N` release, this mode may not work if the video is rotated by 90 or 270 degrees.

Constant Value: 2 (0x00000002)

Public methods

configure

Added in API level 16

```
void configure (MediaFormat format,
                Surface surface,
```

```
MediaCrypto crypto,
int flags)
```

Configures a component.

Parameters	
`format`	`MediaFormat`: The format of the input data (decoder) or the desired format of the output data (encoder). Passing `null` as `format` is equivalent to passing an `an empty mediaformat`.
`surface`	`Surface`: Specify a surface on which to render the output of this decoder. Pass `null` as `surface` if the codec does not generate raw video output (e.g. not a video decoder) and/or if you want to configure the codec for `ByteBuffer` output.
`crypto`	`MediaCrypto`: Specify a crypto object to facilitate secure decryption of the media data. Pass `null` as `crypto` for non-secure codecs.
`flags`	`int`: Specify `CONFIGURE_FLAG_ENCODE` to configure the component as an encoder.

Throws	
`IllegalArgumentException`	if the surface has been released (or is invalid), or the format is unacceptable (e.g. missing a mandatory key), or the flags are not set properly (e.g. missing `CONFIGURE_FLAG_ENCODE` for an encoder).
`IllegalStateException`	if not in the Uninitialized state.
`MediaCodec.CryptoException`	upon DRM error.
`MediaCodec.CodecException`	upon codec error.

createByCodecName

Added in API level 16

```
MediaCodec createByCodecName (String name)
```

If you know the exact name of the component you want to instantiate use this method to instantiate it. Use with caution. Likely to be used with information obtained from `MediaCodecList`

Parameters	
`name`	`String`: The name of the codec to be instantiated.

Returns
`MediaCodec`

createDecoderByType

Added in API level 16

```
MediaCodec createDecoderByType (String type)
```

Instantiate the preferred decoder supporting input data of the given mime type. The following is a partial list of defined mime types and their semantics:

- "video/x-vnd.on2.vp8" - VP8 video (i.e. video in .webm)
- "video/x-vnd.on2.vp9" - VP9 video (i.e. video in .webm)
- "video/avc" - H.264/AVC video
- "video/hevc" - H.265/HEVC video
- "video/mp4v-es" - MPEG4 video
- "video/3gpp" - H.263 video
- "audio/3gpp" - AMR narrowband audio
- "audio/amr-wb" - AMR wideband audio
- "audio/mpeg" - MPEG1/2 audio layer III
- "audio/mp4a-latm" - AAC audio (note, this is raw AAC packets, not packaged in LATM!)
- "audio/vorbis" - vorbis audio
- "audio/g711-alaw" - G.711 alaw audio
- "audio/g711-mlaw" - G.711 ulaw audio

Note: It is preferred to use `findDecoderForFormat(MediaFormat)` and `createByCodecName(String)` to ensure that the resulting codec can handle a given format.

Parameters	
`type`	`String`: The mime type of the input data.

Returns

MediaCodec

createPersistentInputSurface

Added in API level 23

```
Surface createPersistentInputSurface ()
```

Create a persistent input surface that can be used with codecs that normally have an input surface, such as video encoders. A persistent input can be reused by subsequent MediaCodec or MediaRecorder instances, but can only be used by at most one codec or recorder instance concurrently.

The application is responsible for calling release() on the Surface when done.

Returns	
Surface	an input surface that can be used with setInputSurface(Surface).

dequeueInputBuffer

Added in API level 16

```
int dequeueInputBuffer (long timeoutUs)
```

Returns the index of an input buffer to be filled with valid data or -1 if no such buffer is currently available. This method will return immediately if timeoutUs == 0, wait indefinitely for the availability of an input buffer if timeoutUs < 0 or wait up to "timeoutUs" microseconds if timeoutUs > 0.

Parameters	
timeoutUs	long: The timeout in microseconds, a negative timeout indicates "infinite".

Returns
int

Throws	
IllegalStateException	if not in the Executing state, or codec is configured in asynchronous mode.
MediaCodec.CodecException	upon codec error.

dequeueOutputBuffer

Added in API level 16

```
int dequeueOutputBuffer (MediaCodec.BufferInfo info,
                long timeoutUs)
```

Dequeue an output buffer, block at most "timeoutUs" microseconds. Returns the index of an output buffer that has been successfully decoded or one of the INFO_* constants.

Parameters	
info	MediaCodec.BufferInfo: Will be filled with buffer meta data.
timeoutUs	long: The timeout in microseconds, a negative timeout indicates "infinite".

Returns
int

Throws	
IllegalStateException	if not in the Executing state, or codec is configured in asynchronous mode.
MediaCodec.CodecException	upon codec error.

flush

Added in API level 16

```
void flush ()
```

Flush both input and output ports of the component.

Upon return, all indices previously returned in calls to dequeueInputBuffer and dequeueOutputBuffer — or obtained via onInputBufferAvailable or onOutputBufferAvailable callbacks — become invalid, and all buffers are owned by the codec.

If the codec is configured in asynchronous mode, call start() after flush has returned to resume codec operations. The codec will not request input buffers until this has happened. **Note, however, that there may still be outstanding onOutputBufferAvailable callbacks that were not handled prior to calling flush. The indices returned via these callbacks also become invalid upon calling flush and should be discarded.**

If the codec is configured in synchronous mode, codec will resume automatically if it is configured with an input surface. Otherwise, it will resume when `dequeueInputBuffer` is called.

Throws	
`IllegalStateException`	if not in the Executing state.
`MediaCodec.CodecException`	upon codec error.

getCodecInfo

Added in API level 18

```
MediaCodecInfo getCodecInfo ()
```

Get the codec info. If the codec was created by createDecoderByType or createEncoderByType, what component is chosen is not known beforehand, and thus the caller does not have the MediaCodecInfo.

Returns
`MediaCodecInfo`

Throws	
`IllegalStateException`	if in the Released state.

getInputBuffer

Added in API level 21

```
ByteBuffer getInputBuffer (int index)
```

Returns a `cleared`, writable ByteBuffer object for a dequeued input buffer index to contain the input data. After calling this method any ByteBuffer or Image object previously returned for the same input index MUST no longer be used.

Parameters	
`index`	`int`: The index of a client-owned input buffer previously returned from a call to `dequeueInputBuffer(long)`, or received via an onInputBufferAvailable callback.

Returns	
`ByteBuffer`	the input buffer, or null if the index is not a dequeued input buffer, or if the codec is configured for surface input.

Throws	
`IllegalStateException`	if not in the Executing state.
`MediaCodec.CodecException`	upon codec error.

getInputBuffers

Added in API level 16

```
ByteBuffer[] getInputBuffers ()
```

This method was deprecated in API level 21.
Use the new `getInputBuffer(int)` method instead each time an input buffer is dequeued. **Note:** As of API 21, dequeued input buffers are automatically `cleared`. *Do not use this method if using an input surface.*

Retrieve the set of input buffers. Call this after start() returns. After calling this method, any ByteBuffers previously returned by an earlier call to this method MUST no longer be used.

Returns
`ByteBuffer[]`

Throws	
`IllegalStateException`	if not in the Executing state, or codec is configured in asynchronous mode.
`MediaCodec.CodecException`	upon codec error.

getInputFormat

Added in API level 21

```
MediaFormat getInputFormat ()
```

Call this after `configure(MediaFormat, Surface, MediaCrypto, int)` returns successfully to get the input format accepted by the codec. Do this to determine what optional configuration parameters were supported by the codec.

Returns
MediaFormat

Throws	
IllegalStateException	if not in the Executing or Configured state.
MediaCodec.CodecException	upon codec error.

getInputImage

Added in API level 21

```
Image getInputImage (int index)
```

Returns a writable Image object for a dequeued input buffer index to contain the raw input video frame. After calling this method any ByteBuffer or Image object previously returned for the same input index MUST no longer be used.

Parameters	
index	int: The index of a client-owned input buffer previously returned from a call to dequeueInputBuffer(long), or received via an onInputBufferAvailable callback.

Returns	
Image	the input image, or null if the index is not a dequeued input buffer, or not a ByteBuffer that contains a raw image.

Throws	
IllegalStateException	if not in the Executing state.
MediaCodec.CodecException	upon codec error.

getName

Added in API level 18

```
String getName ()
```

Get the component name. If the codec was created by createDecoderByType or createEncoderByType, what component is chosen is not known beforehand.

Returns
String

Throws	
IllegalStateException	if in the Released state.

getOutputBuffer

Added in API level 21

```
ByteBuffer getOutputBuffer (int index)
```

Returns a read-only ByteBuffer for a dequeued output buffer index. The position and limit of the returned buffer are set to the valid output data. After calling this method, any ByteBuffer or Image object previously returned for the same output index MUST no longer be used.

Parameters	
index	int: The index of a client-owned output buffer previously returned from a call to dequeueOutputBuffer(MediaCodec.BufferInfo, long), or received via an onOutputBufferAvailable callback.

Returns	
ByteBuffer	the output buffer, or null if the index is not a dequeued output buffer, or the codec is configured with an output surface.

Throws	
IllegalStateException	if not in the Executing state.
MediaCodec.CodecException	upon codec error.

getOutputBuffers

Added in API level 16

`ByteBuffer[] getOutputBuffers ()`

This method was deprecated in API level 21.
Use the new `getOutputBuffer(int)` method instead each time an output buffer is dequeued. This method is not supported if codec is configured in asynchronous mode. **Note:** As of API 21, the position and limit of output buffers that are dequeued will be set to the valid data range. *Do not use this method if using an output surface.*

Retrieve the set of output buffers. Call this after start() returns and whenever dequeueOutputBuffer signals an output buffer change by returning `INFO_OUTPUT_BUFFERS_CHANGED`. After calling this method, any ByteBuffers previously returned by an earlier call to this method MUST no longer be used.

Returns
`ByteBuffer[]`

Throws	
`IllegalStateException`	if not in the Executing state, or codec is configured in asynchronous mode.
`MediaCodec.CodecException`	upon codec error.

getOutputFormat

Added in API level 21

`MediaFormat getOutputFormat (int index)`

Returns the output format for a specific output buffer.

Parameters	
index	`int`: The index of a client-owned input buffer previously returned from a call to `dequeueInputBuffer(long)`.

Returns	
`MediaFormat`	the format for the output buffer, or null if the index is not a dequeued output buffer.

getOutputFormat

Added in API level 16

`MediaFormat getOutputFormat ()`

Call this after dequeueOutputBuffer signals a format change by returning `INFO_OUTPUT_FORMAT_CHANGED`. You can also call this after `configure(MediaFormat, Surface, MediaCrypto, int)` returns successfully to get the output format initially configured for the codec. Do this to determine what optional configuration parameters were supported by the codec.

Returns
`MediaFormat`

Throws	
`IllegalStateException`	if not in the Executing or Configured state.
`MediaCodec.CodecException`	upon codec error.

getOutputImage

Added in API level 21

`Image getOutputImage (int index)`

Returns a read-only Image object for a dequeued output buffer index that contains the raw video frame. After calling this method, any ByteBuffer or Image object previously returned for the same output index MUST no longer be used.

Parameters	
index	`int`: The index of a client-owned output buffer previously returned from a call to `dequeueOutputBuffer(MediaCodec.BufferInfo, long)`, or received via an onOutputBufferAvailable callback.

Returns	
`Image`	the output image, or null if the index is not a dequeued output buffer, not a raw video frame, or if the codec was configured with an output surface.

Throws	
`IllegalStateException`	if not in the Executing state.

`MediaCodec.CodecException`	upon codec error.

queueInputBuffer

Added in API level 16

```
void queueInputBuffer (int index,
                int offset,
                int size,
                long presentationTimeUs,
                int flags)
```

After filling a range of the input buffer at the specified index submit it to the component. Once an input buffer is queued to the codec, it MUST NOT be used until it is later retrieved by `getInputBuffer(int)` in response to a `dequeueInputBuffer(long)` return value or a `onInputBufferAvailable(MediaCodec, int)` callback.

Many decoders require the actual compressed data stream to be preceded by "codec specific data", i.e. setup data used to initialize the codec such as PPS/SPS in the case of AVC video or code tables in the case of vorbis audio. The class `MediaExtractor` provides codec specific data as part of the returned track format in entries named "csd-0", "csd-1" ...

These buffers can be submitted directly after `start()` or `flush()` by specifying the flag `BUFFER_FLAG_CODEC_CONFIG`. However, if you configure the codec with a `MediaFormat` containing these keys, they will be automatically submitted by MediaCodec directly after start. Therefore, the use of `BUFFER_FLAG_CODEC_CONFIG` flag is discouraged and is recommended only for advanced users.

To indicate that this is the final piece of input data (or rather that no more input data follows unless the decoder is subsequently flushed) specify the flag `BUFFER_FLAG_END_OF_STREAM`.

Note: Prior to `M`, `presentationTimeUs` was not propagated to the frame timestamp of (rendered) Surface output buffers, and the resulting frame timestamp was undefined. Use `releaseOutputBuffer(int, long)` to ensure a specific frame timestamp is set. Similarly, since frame timestamps can be used by the destination surface for rendering synchronization, **care must be taken to normalize presentationTimeUs so as to not be mistaken for a system time. (See SurfaceView specifics).**

Parameters

`index`	`int`: The index of a client-owned input buffer previously returned in a call to `dequeueInputBuffer(long)`.
`offset`	`int`: The byte offset into the input buffer at which the data starts.
`size`	`int`: The number of bytes of valid input data.
`presentationTimeUs`	`long`: The presentation timestamp in microseconds for this buffer. This is normally the media time at which this buffer should be presented (rendered). When using an output surface, this will be propagated as the `timestamp` for the frame (after conversion to nanoseconds).
`flags`	`int`: A bitmask of flags `BUFFER_FLAG_CODEC_CONFIG` and `BUFFER_FLAG_END_OF_STREAM`. While not prohibited, most codecs do not use the `BUFFER_FLAG_KEY_FRAME` flag for input buffers.

queueSecureInputBuffer

Added in API level 16

```
void queueSecureInputBuffer (int index,
                int offset,
                MediaCodec.CryptoInfo info,
                long presentationTimeUs,
                int flags)
```

Similar to `queueInputBuffer` but submits a buffer that is potentially encrypted. **Check out further notes at `queueInputBuffer`.**

Parameters

`index`	`int`: The index of a client-owned input buffer previously returned in a call to `dequeueInputBuffer(long)`.
`offset`	`int`: The byte offset into the input buffer at which the data starts.
`info`	`MediaCodec.CryptoInfo`: Metadata required to facilitate decryption, the object can be reused immediately after this call returns.
`presentationTimeUs`	`long`: The presentation timestamp in microseconds for this buffer. This is normally the media time at which this buffer should be presented (rendered).
`flags`	`int`: A bitmask of flags `BUFFER_FLAG_CODEC_CONFIG` and `BUFFER_FLAG_END_OF_STREAM`. While not prohibited, most codecs do not use the `BUFFER_FLAG_KEY_FRAME` flag for input buffers.

release

Added in API level 16

```
void release ()
```

Free up resources used by the codec instance. Make sure you call this when you're done to free up any opened component instance instead of relying on the garbage collector to do this for you at some point in the future.

releaseOutputBuffer

Added in API level 16

```
void releaseOutputBuffer (int index,
                boolean render)
```

If you are done with a buffer, use this call to return the buffer to the codec or to render it on the output surface. If you configured the codec with an output surface, setting `render` to `true` will first send the buffer to that output surface. The surface will release the buffer back to the codec once it is no longer used/displayed. Once an output buffer is released to the codec, it MUST NOT be used until it is later retrieved by `getOutputBuffer(int)` in response to a `dequeueOutputBuffer(MediaCodec.BufferInfo, long)` return value or a `onOutputBufferAvailable(MediaCodec, int, MediaCodec.BufferInfo)` callback.

Parameters

`index`	`int`: The index of a client-owned output buffer previously returned from a call to `dequeueOutputBuffer(MediaCodec.BufferInfo, long)`.
`render`	`boolean`: If a valid surface was specified when configuring the codec, passing true renders this output buffer to the surface.

Throws

`IllegalStateException`	if not in the Executing state.
`MediaCodec.CodecException`	upon codec error.

releaseOutputBuffer

Added in API level 21

```
void releaseOutputBuffer (int index,
                long renderTimestampNs)
```

If you are done with a buffer, use this call to update its surface timestamp and return it to the codec to render it on the output surface. If you have not specified an output surface when configuring this video codec, this call will simply return the buffer to the codec.

The timestamp may have special meaning depending on the destination surface.

SurfaceView specifics

If you render your buffer on a `SurfaceView`, you can use the timestamp to render the buffer at a specific time (at the VSYNC at or after the buffer timestamp). For this to work, the timestamp needs to be *reasonably close* to the current `nanoTime()`. Currently, this is set as within one (1) second. A few notes:

- the buffer will not be returned to the codec until the timestamp has passed and the buffer is no longer used by the `Surface`.
- buffers are processed sequentially, so you may block subsequent buffers to be displayed on the `Surface`. This is important if you want to react to user action, e.g. stop the video or seek.
- if multiple buffers are sent to the `Surface` to be rendered at the same VSYNC, the last one will be shown, and the other ones will be dropped.
- if the timestamp is *not* "reasonably close" to the current system time, the `Surface` will ignore the timestamp, and display the buffer at the earliest feasible time. In this mode it will not drop frames.
- for best performance and quality, call this method when you are about two VSYNCs' time before the desired render time. For 60Hz displays, this is about 33 msec.

Once an output buffer is released to the codec, it MUST NOT be used until it is later retrieved by `getOutputBuffer(int)` in response to a `dequeueOutputBuffer(MediaCodec.BufferInfo, long)` return value or a `onOutputBufferAvailable(MediaCodec, int, MediaCodec.BufferInfo)` callback.

Parameters

`index`	`int`: The index of a client-owned output buffer previously returned from a call to `dequeueOutputBuffer(MediaCodec.BufferInfo, long)`.
`renderTimestampNs`	`long`: The timestamp to associate with this buffer when it is sent to the Surface.

Throws

`IllegalStateException`	if not in the Executing state.
`MediaCodec.CodecException`	upon codec error.

setCallback

Added in API level 23

```
void setCallback (MediaCodec.Callback cb,
                Handler handler)
```

Sets an asynchronous callback for actionable MediaCodec events. If the client intends to use the component in asynchronous mode, a valid callback should be provided before `configure(MediaFormat, Surface, MediaCrypto, int)` is called. When asynchronous callback is enabled, the client should not call `getInputBuffers()`, `getOutputBuffers()`, `dequeueInputBuffer(long)` or `dequeueOutputBuffer(BufferInfo, long)`.

Also, `flush()` behaves differently in asynchronous mode. After calling `flush`, you must call `start()` to "resume" receiving input buffers, even if an input surface was created.

Parameters

`cb`	`MediaCodec.Callback`: The callback that will run. Use `null` to clear a previously set callback (before `configure` is called and run in synchronous mode).

handler	Handler: Callbacks will happen on the handler's thread. If null, callbacks are done on the default thread (the caller's thread or the main thread.)

setOnFrameRenderedListener

Added in API level 23

```
void setOnFrameRenderedListener (MediaCodec.OnFrameRenderedListener listener,
                Handler handler)
```

Registers a callback to be invoked when an output frame is rendered on the output surface.

This method can be called in any codec state, but will only have an effect in the Executing state for codecs that render buffers to the output surface.

Note: This callback is for informational purposes only: to get precise render timing samples, and can be significantly delayed and batched. Some frames may have been rendered even if there was no callback generated.

Parameters

listener	MediaCodec.OnFrameRenderedListener: the callback that will be run
handler	Handler: the callback will be run on the handler's thread. If null, the callback will be run on the default thread, which is the looper from which the codec was created, or a new thread if there was none.

setOutputSurface

Added in API level 23

```
void setOutputSurface (Surface surface)
```

Dynamically sets the output surface of a codec.

This can only be used if the codec was configured with an output surface. The new output surface should have a compatible usage type to the original output surface. E.g. codecs may not support switching from a SurfaceTexture (GPU readable) output to ImageReader (software readable) output.

Parameters

surface	Surface: the output surface to use. It must not be null.

Throws

IllegalStateException	if the codec does not support setting the output surface in the current state.
IllegalArgumentException	if the new surface is not of a suitable type for the codec.

setParameters

Added in API level 19

```
void setParameters (Bundle params)
```

Communicate additional parameter changes to the component instance. **Note:** Some of these parameter changes may silently fail to apply.

Parameters

params	Bundle: The bundle of parameters to set.

Throws

IllegalStateException	if in the Released state.

setVideoScalingMode

Added in API level 16

```
void setVideoScalingMode (int mode)
```

If a surface has been specified in a previous call to configure(MediaFormat, Surface, MediaCrypto, int) specifies the scaling mode to use. The default is "scale to fit".

The scaling mode may be reset to the **default** each time an INFO_OUTPUT_BUFFERS_CHANGED event is received from the codec; therefore, the client must call this method after every buffer change event (and before the first output buffer is released for rendering) to ensure consistent scaling mode.

Since the INFO_OUTPUT_BUFFERS_CHANGED event is deprecated, this can also be done after each INFO_OUTPUT_FORMAT_CHANGED event.

Parameters

mode	int

Throws

IllegalArgumentException	if mode is not recognized.
IllegalStateException	if in the Released state.

start

Added in API level 16

```
void start ()
```

After successfully configuring the component, call `start`.

Call `start` also if the codec is configured in asynchronous mode, and it has just been flushed, to resume requesting input buffers.

stop

Added in API level 16

```
void stop ()
```

Finish the decode/encode session, note that the codec instance remains active and ready to be `start()`ed again. To ensure that it is available to other client call `release()` and don't just rely on garbage collection to eventually do this for you.

Throws	
IllegalStateException	if in the Released state.

Protected methods

finalize

Added in API level 16

```
void finalize ()
```

Called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the `finalize` method to dispose of system resources or to perform other cleanup.

The general contract of `finalize` is that it is invoked if and when the Java™ virtual machine has determined that there is no longer any means by which this object can be accessed by any thread that has not yet died, except as a result of an action taken by the finalization of some other object or class which is ready to be finalized. The `finalize` method may take any action, including making this object available again to other threads; the usual purpose of `finalize`, however, is to perform cleanup actions before the object is irrevocably discarded. For example, the finalize method for an object that represents an input/output connection might perform explicit I/O transactions to break the connection before the object is permanently discarded.

The `finalize` method of class `Object` performs no special action; it simply returns normally. Subclasses of `Object` may override this definition.

The Java programming language does not guarantee which thread will invoke the `finalize` method for any given object. It is guaranteed, however, that the thread that invokes finalize will not be holding any user-visible synchronization locks when finalize is invoked. If an uncaught exception is thrown by the finalize method, the exception is ignored and finalization of that object terminates.

After the `finalize` method has been invoked for an object, no further action is taken until the Java virtual machine has again determined that there is no longer any means by which this object can be accessed by any thread that has not yet died, including possible actions by other objects or classes which are ready to be finalized, at which point the object may be discarded.

The `finalize` method is never invoked more than once by a Java virtual machine for any given object.

Any exception thrown by the `finalize` method causes the finalization of this object to be halted, but is otherwise ignored.