Basic photo, video, and audio capture with MediaCapture

02/08/2017 • 12 minutes to read • +5

In this article

This article shows the simplest way to capture photos and video using the **MediaCapture** class. The **MediaCapture** class exposes a robust set of APIs that provide low-level control over the capture pipeline and enable advanced capture scenarios, but this article is intended to help you add basic media capture to your app quickly and easily. To learn about more of the features that **MediaCapture** provides, see **Camera**.

If you simply want to capture a photo or video and don't intend to add any additional media capture features, or if you don't want to create your own camera UI, you may want to use the **CameraCaptureUI** class, which allows you to simply launch the Windows built-in camera app and receive the photo or video file that was captured. For more information, see **Capture photos and video with Windows built-in camera UI**

The code in this article was adapted from the **Camera starter kit** sample. You can download the sample to see the code used in context or to use the sample as a starting point for your own app.

Add capability declarations to the app manifest

In order for your app to access a device's camera, you must declare that your app uses the *webcam* and *microphone* device capabilities. If you want to save captured photos and videos to the users's Pictures or Videos library, you must also declare the *picturesLibrary* and *videosLibrary* capability.

To add capabilities to the app manifest

1. In Microsoft Visual Studio, in **Solution Explorer**, open the designer for the application manifest by double-clicking the **package.appxmanifest** item.
2. Select the **Capabilities** tab.
3. Check the box for **Webcam** and the box for **Microphone**.
4. For access to the Pictures and Videos library check the boxes for **Pictures Library** and the box for **Videos Library**.

Initialize the MediaCapture object

All of the capture methods described in this article require the first step of initializing the MediaCapture object by calling the constructor and then calling InitializeAsync. Since the **MediaCapture** object will be accessed from multiple places in your app, declare a class variable to hold the object. Implement a handler for the **MediaCapture** object's Failed event to be notified if a capture operation fails.

C# Copy

```csharp
MediaCapture mediaCapture;
bool isPreviewing;
```

C# Copy

```csharp
mediaCapture = new MediaCapture();
await mediaCapture.InitializeAsync();
mediaCapture.Failed += MediaCapture_Failed;
```

Set up the camera preview

It's possible to capture photos, videos, and audio using **MediaCapture** without showing the camera preview, but typically you want to show the preview stream so that the user can see what's being captured. Also, a few **MediaCapture** features require the preview stream to be running before they can be enbled, including auto focus, auto exposure, and auto white balance. To see how to set up the camera preview, see Display the camera preview.

Capture a photo to a SoftwareBitmap

The SoftwareBitmap class was introduced in Windows 10 to provide a common representation of images across multiple features. If you want to capture a photo and then immediately use the captured image in your app, such as displaying it in XAML, instead of capturing to a file, then you should capture to a **SoftwareBitmap**. You still have the option of saving the image to disk later.

After initializing the **MediaCapture** object, you can capture a photo to a **SoftwareBitmap** using the **LowLagPhotoCapture** class. Get an instance of this class by calling **PrepareLowLagPhotoCaptureAsync**, passing in an **ImageEncodingProperties** object specifying the image format you want. **CreateUncompressed** creates an uncompressed encoding with the specified pixel format. Capture a photo by calling **CaptureAsync**, which returns a **CapturedPhoto** object. Get a **SoftwareBitmap** by accessing the **Frame** property and then the **SoftwareBitmap** property.

If you want, you can capture multiple photos by repeatedly calling **CaptureAsync**. When you are done capturing, call **FinishAsync** to shut down the **LowLagPhotoCapture** session and free up the associated resources. After calling **FinishAsync**, to begin capturing photos again you will need to call **PrepareLowLagPhotoCaptureAsync** again to reinitialize the capture session before calling **CaptureAsync**.

C# Copy

```
// Prepare and capture photo
var lowLagCapture = await
mediaCapture.PrepareLowLagPhotoCaptureAsync(ImageEncodingProperties.Creat
eUncompressed(MediaPixelFormat.Bgra8));

var capturedPhoto = await lowLagCapture.CaptureAsync();
var softwareBitmap = capturedPhoto.Frame.SoftwareBitmap;

await lowLagCapture.FinishAsync();
```

Starting with Windows, version 1803, you can access the **BitmapProperties** property of the **CapturedFrame** class returned from **CaptureAsync** to retrieve metadata about the captured photo. You can pass this data into a **BitmapEncoder** to save the metadata to a file. Previously, there was no way to access this data for uncompressed image formats. You can also access the **ControlValues** property to retrieve a **CapturedFrameControlValues** object that describes the control values, such as exposure and white balance, for the captured frame.

For information about using **BitmapEncoder** and about working with the **SoftwareBitmap** object, including how to display one in a XAML page, see **Create, edit, and save bitmap images**.

For more information on setting capture device control values, see Capture device controls for photo and video.

Starting with Windows 10, version 1803, you can get the metadata, such as EXIF information, for photos captured in uncompressed format by accessing the **BitmapProperties** property of the **CapturedFrame** returned by **MediaCapture**. In previous releases this data was only accessible in the header of photos captured to a compressed file format. You can provide this data to a **BitmapEncoder** when manually writing an image file. For more information on encoding bitmaps, see Create, edit, and save bitmap images. You can also access the frame control values, such as exposure and flash settings, used when the image was captured by accessing the **ControlValues** property. For more information, see Capture device controls for photo and video capture.

Capture a photo to a file

A typical photography app will save a captured photo to disk or to cloud storage and will need to add metadata, such as photo orientation, to the file. The following example shows you how to capture an photo to a file. You still have the option of creating a **SoftwareBitmap** from the image file later.

The technique shown in this example captures the photo to an in-memory stream and then transcode the photo from the stream to a file on disk. This example uses **GetLibraryAsync** to get the user's pictures library and then the **SaveFolder** property to get a reference default save folder. Remember to add the **Pictures Library** capability to your app manifest to access this folder. **CreateFileAsync** creates a new **StorageFile** to which the photo will be saved.

Create an **InMemoryRandomAccessStream** and then call **CapturePhotoToStreamAsync** to capture a photo to the stream, passing in the stream and an **ImageEncodingProperties** object specifying the image format that should be used. You can create custom encoding properties by initializing the object yourself, but the class provides static methods, like **ImageEncodingProperties.CreateJpeg** for common encoding formats. Next, create a file stream to the output file by calling **OpenAsync**. Create a **BitmapDecoder** to decode the image from the in memory stream and then create a **BitmapEncoder** to encode the image to file by calling **CreateForTranscodingAsync**.

You can optionally create a **BitmapPropertySet** object and then call **SetPropertiesAsync** on the image encoder to include metadata about the photo in the image file. For more information about encoding properties, see **Image metadata**. Handling device orientation properly is essential for most photography apps. For more information, see **Handle device orientation with MediaCapture**.

Finally, call **FlushAsync** on the encoder object to transcode the photo from the in-memory stream to the file.

C# Copy

```
var myPictures = await
Windows.Storage.StorageLibrary.GetLibraryAsync(Windows.Storage.KnownLibra
ryId.Pictures);
StorageFile file = await
myPictures.SaveFolder.CreateFileAsync("photo.jpg",
CreationCollisionOption.GenerateUniqueName);

using (var captureStream = new InMemoryRandomAccessStream())
{
    await
mediaCapture.CapturePhotoToStreamAsync(ImageEncodingProperties.CreateJpeg
(), captureStream);

    using (var fileStream = await
file.OpenAsync(FileAccessMode.ReadWrite))
    {
        var decoder = await BitmapDecoder.CreateAsync(captureStream);
        var encoder = await
BitmapEncoder.CreateForTranscodingAsync(fileStream, decoder);

        var properties = new BitmapPropertySet {
            { "System.Photo.Orientation", new
BitmapTypedValue(PhotoOrientation.Normal, PropertyType.UInt16) }
        };
        await encoder.BitmapProperties.SetPropertiesAsync(properties);

        await encoder.FlushAsync();
    }
}
```

For more information about working with files and folders, see Files, folders, and libraries.

Capture a video

Quickly add video capture to your app by using the LowLagMediaRecording class. First, declare a class variable to for the object.

C# Copy

```
LowLagMediaRecording _mediaRecording;
```

Next, create a **StorageFile** object to which the video will be saved. Note that to save to the user's video library, as shown in this example, you must add the **Videos Library** capability to your app manifest. Call PrepareLowLagRecordToStorageFileAsync to initialize the media recording, passing in the storage file and a MediaEncodingProfile object specifying the encoding for the video. The class provides static methods, like CreateMp4, for creating common video encoding profiles.

Finally, call StartAsync to begin capturing video.

C# Copy

```
var myVideos = await
Windows.Storage.StorageLibrary.GetLibraryAsync(Windows.Storage.KnownLibra
ryId.Videos);
StorageFile file = await myVideos.SaveFolder.CreateFileAsync("video.mp4",
CreationCollisionOption.GenerateUniqueName);
_mediaRecording = await
mediaCapture.PrepareLowLagRecordToStorageFileAsync(
        MediaEncodingProfile.CreateMp4(VideoEncodingQuality.Auto), file);
await _mediaRecording.StartAsync();
```

To stop recording video, call StopAsync.

C# Copy

```
await _mediaRecording.StopAsync();
```

You can continue to call **StartAsync** and **StopAsync** to capture additional videos. When you are done capturing videos, call FinishAsync to dispose of the capture session and clean up associated resources. After this call, you must call **PrepareLowLagRecordToStorageFileAsync** again to reinitialize the capture session before calling **StartAsync**.

C# Copy

```
await _mediaRecording.FinishAsync();
```

When capturing video, you should register a handler for the RecordLimitationExceeded event of the **MediaCapture** object, which will be raised by the operating system if you surpass the limit for a single recording, currently three hours. In the handler for the event, you should finalize your recording by calling StopAsync.

C# Copy

```
mediaCapture.RecordLimitationExceeded +=
MediaCapture_RecordLimitationExceeded;
```

C# Copy

```
private async void MediaCapture_RecordLimitationExceeded(MediaCapture
sender)
{
    await _mediaRecording.StopAsync();
    System.Diagnostics.Debug.WriteLine("Record limitation exceeded.");
}
```

Play and edit captured video files

Once you have captured a video to a file, you may want to load the file and play it back within your app's UI. You can do this using the **MediaPlayerElement** XAML control and an associated **MediaPlayer**. For information on playing media in a XAML page, see Play audio and video with MediaPlayer.

You can also create a **MediaClip** object from a video file by calling **CreateFromFileAsync**. A **MediaComposition** provides basic video editing functionality like arranging the sequence of **MediaClip** objects, trimming video length, creating layers, adding background music, and applying video effects. For more information on working with media compositions, see Media compositions and editing.

Pause and resume video recording

You can pause a video recording and then resume recording without creating a separate output file by calling **PauseAsync** and then calling **ResumeAsync**.

C# Copy

```
await
_mediaRecording.PauseAsync(Windows.Media.Devices.MediaCapturePauseBehavio
r.ReleaseHardwareResources);
```

C# Copy

```
await _mediaRecording.ResumeAsync();
```

Starting with Windows 10, version 1607, you can pause a video recording and receive the last frame captured before the recording was paused. You can then overlay this frame on the camera preview to allow the user to align the camera with the paused frame before resuming recording. Calling **PauseWithResultAsync** returns a **MediaCapturePauseResult** object. The **LastFrame** property is a **VideoFrame** object representing the last frame. To display the frame in XAML, get the **SoftwareBitmap** representation of the video frame. Currently, only images in BGRA8 format with premultiplied or empty alpha channel are supported, so call Convert if necessary to get the correct format. Create a new SoftwareBitmapSource object and call SetBitmapAsync to initialize it. Finally, set the **Source** property of a XAML Image control to display the image. For this trick to work, your image must be aligned with the **CaptureElement** control and should have an opacity value less than one. Don't forget that you can only modify the UI on the UI thread, so make this call inside RunAsync.

PauseWithResultAsync also returns the duration of the video that was recorded in the preceeding segment in case you need to track how much total time has been recorded.

C# Copy

```
MediaCapturePauseResult result = 
    await
_mediaRecording.PauseWithResultAsync(Windows.Media.Devices.MediaCapturePa
useBehavior.RetainHardwareResources);

var pausedFrame = result.LastFrame.SoftwareBitmap;
if(pausedFrame.BitmapPixelFormat != BitmapPixelFormat.Bgra8 ||
pausedFrame.BitmapAlphaMode != BitmapAlphaMode.Ignore)
{
    pausedFrame = SoftwareBitmap.Convert(pausedFrame,
BitmapPixelFormat.Bgra8, BitmapAlphaMode.Ignore);
}

var source = new SoftwareBitmapSource();
await source.SetBitmapAsync(pausedFrame);

await Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
{
    PauseImage.Source = source;
    PauseImage.Visibility = Visibility.Visible;
});

_totalRecordedTime += result.RecordDuration;
```

When you resume recording, you can set the source of the image to null and hide it.

C# Copy

```
await Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
{
    PauseImage.Source = null;
    PauseImage.Visibility = Visibility.Collapsed;
});

await _mediaRecording.ResumeAsync();
```

Note that you can also get a result frame when you stop the video by calling StopWithResultAsync.

Capture audio

You can quickly add audio capture to your app by using the same technique shown above for capturing video. The example below creates a **StorageFile** in the application data folder. Call PrepareLowLagRecordToStorageFileAsync to initialize the capture session, passing in the file and a MediaEncodingProfile which is generated in this example by the CreateMp3 static method. To begin recording, call StartAsync.

C# Copy

```
mediaCapture.RecordLimitationExceeded +=
MediaCapture_RecordLimitationExceeded;

var localFolder = Windows.Storage.ApplicationData.Current.LocalFolder;
StorageFile file = await localFolder.CreateFileAsync("audio.mp3",
CreationCollisionOption.GenerateUniqueName);
_mediaRecording = await
mediaCapture.PrepareLowLagRecordToStorageFileAsync(
        MediaEncodingProfile.CreateMp3(AudioEncodingQuality.High), file);
await _mediaRecording.StartAsync();
```

Call StopAsync to stop the audio recording.

Related topics

- Camera

C# Copy

```
await _mediaRecording.StopAsync();
```

You can call **StartAsync** and **StopAsync** multiple times to record several audio files. When you are done capturing audio, call FinishAsync to dispose of the capture session and clean up associated resources. After this call, you must call **PrepareLowLagRecordToStorageFileAsync** again to reinitialize the capture session before calling **StartAsync**.

C# Copy

```
await _mediaRecording.FinishAsync();
```

Detect and respond to audio level changes by the system

Starting with Windows 10, version 1803, your app can detect when the system lowers or mutes the audio level of your app's audio capture and audio render streams. For example, the system may mute your app's streams when it goes into the background. The AudioStateMonitor class allows you to register to receive an event when the system modifies the volume of an audio stream. Get an instance of **AudioStateMonitor** for monitoring audio capture streams by calling CreateForCaptureMonitoring. Get an instance for monitoring audio render streams by calling CreateForRenderMonitoring. Register a handler for the SoundLevelChanged event of each monitor to be notified when the audio for the corresponding stream category is changed by the system.

C# Copy

```
// Namespaces for monitoring audio state
using Windows.Media;
using Windows.Media.Audio;
```

C# Copy

```
AudioStateMonitor captureAudioStateMonitor;
AudioStateMonitor renderAudioStateMonitor;
```

C# Copy

```
captureAudioStateMonitor =
AudioStateMonitor.CreateForCaptureMonitoring();
captureAudioStateMonitor.SoundLevelChanged +=
CaptureAudioStateMonitor_SoundLevelChanged; ;

renderAudioStateMonitor = AudioStateMonitor.CreateForRenderMonitoring();
renderAudioStateMonitor.SoundLevelChanged +=
RenderAudioStateMonitor_SoundLevelChanged; ;
```

In the **SoundLevelChanged** handler for the capture stream, you can check the SoundLevel property of the **AudioStateMonitor** sender to determine the new sound level. Note that a capture stream should never be lowered, or "ducked", by the system. It should only ever be muted or switched back to full volume. If the audio stream is muted, you can stop a capture in progress. If the audio stream is restored to full volume, you can start capturing again. The following example uses some boolean class variables to track whether the app is currently capturing audio and if the capture was stopped due to the audio state change. These variables are used to determine when it's appropriate to programmatically stop or start audio capture.

C# Copy

```
bool isCapturingAudio = false;
bool capturingStoppedForAudioState = false;
```

```
private void CaptureAudioStateMonitor_SoundLevelChanged(AudioStateMonitor
sender, object args)
{
    switch (sender.SoundLevel)
    {
        case SoundLevel.Full:
            if(capturingStoppedForAudioState)
            {
                StartAudioCapture();
                capturingStoppedForAudioState = false;
            }
            break;
        case SoundLevel.Muted:
            if(isCapturingAudio)
            {
                StopAudioCapture();
                capturingStoppedForAudioState = true;
            }
            break;
        case SoundLevel.Low:
            // This should never happen for capture
            Debug.WriteLine("Unexpected audio state.");
            break;
    }
}
```

The following code example illustrates an implementation of the **SoundLevelChanged** handler for audio rendering. Depending on your app scenario, and the type of content you are playing, you may want to pause audio playback when the sound level is ducked. For more information on handling sound level changes for media playback, see Play audio and video with MediaPlayer.

C# Copy

```
private void RenderAudioStateMonitor_SoundLevelChanged(AudioStateMonitor
sender, object args)
{
    if ((sender.SoundLevel == SoundLevel.Full) ||
  (sender.SoundLevel == SoundLevel.Low && !isPodcast))
    {
        mediaPlayer.Play();
    }
    else if ((sender.SoundLevel == SoundLevel.Muted) ||
         (sender.SoundLevel == SoundLevel.Low && isPodcast))
    {
        // Pause playback if we’re muted or if we’re playing a podcast
and are ducked
        mediaPlayer.Pause();
    }
}
```

- Capture photos and video with Windows built-in camera UI
- Handle device orientation with MediaCapture

- Create, edit, and save bitmap images
- Files, folders, and libraries

Is this page helpful?

Yes No