Using a Camera with Windows Mobile 5

By Amit Ranjan | July 21, 2006

A number of Windows Mobile 5.0 APIs (for example, SHCameraCapture) make it trivial for a mobile application developer to access a camera, but their ease of use comes at a price—flexibility. Most of the time, using the API directly would offer a solution, but sometimes you need more control and flexibility. That's where Microsoft's DirectShow framework comes in. This article shows how to use DirectShow to access a camera. It demonstrates how to build a filter graph manually and how to handle graph events in the application message handler. Having some prior knowledge of DirectShow and COM will be helpful, but it's not necessary.

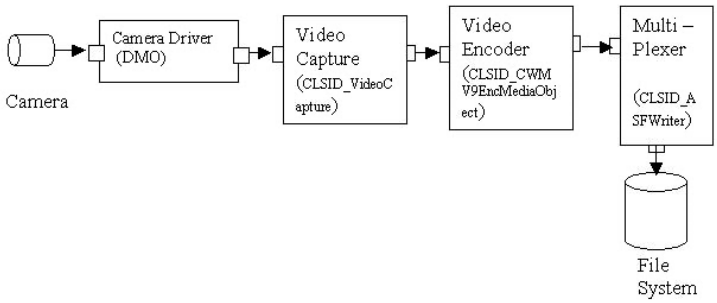


Figure 1: Filter Graph for Video Capture

The camera is the hardware component. For an application to interact with the camera, it would need to talk to its drivers. Next, the video capture filter enables an application to capture video. After capture, you encode the data using WMV9EncMediaObject, a DirectX Media Object (DMO). You can use a DMO inside a filter graph with the help of a DMO Wrapper filter. Next, the encoded video data needs to be multiplexed. You use a Windows Media ASF writer filter for this task. The ASF writer multiplexes the video data and writes it to an .asf file. With that, your filter graph is ready. Now, it's just a matter of running it. As you will see, building the graph is pretty easy too.

Set the Build Environment

First, you need to set the build environment. Add the following libraries in the linker setting of a Visual Studio 2005 Smart Device project:

- dmoguids.lib
- strmiids.lib
- strmbase.lib
- uuid.lib

Also include the following header files in your project:

- atlbase.h
- dmodshow.h
- dmoreg.h
- wmcodecids.h

Note: *For the sake of clarity, this example doesn't show error handling. However, a real world application would require error handling.*

Building the Graph

A filter graph that performs audio or video capture is known as a Capture graph. DirectShow provides a Capture Graph Builder object that exposes an interface called ICaptureGraphBuilder2; it exposes methods to help build and control a capture graph.

First, create instances of IGraphBuilder and ICaptureGraphBuilder2 by using the COM function CoCreateInstance:

```
HRESULT hResult = S_OK;

IGraphBuilder *pFilterGraph;
```

```
ICaptureGraphBuilder2 *pCaptureGraphBuil

hResult=CoCreateInstance(CLSID_FilterGra

                         IID_IGraphBuild

hResult=CoCreateInstance(CLSID_CaptureGr

                         CLSCTX_INPROC,

                         (void**)& pCapt
```

CoCreateInstance takes five parameters:

1. The first is a class ID.
2. The second decides whether the object created is part of an aggregator.
3. The third specifies the context in which the newly created object would run.
4. The fourth parameter is a reference to the identifier of the interface you will use to communicate with the object.
5. The last parameter is the address of the variable that receives the interface pointer requested.

Once you have created the IGraphBuilder and ICaptureGraphBulder2 instances, you need to

call the SetFilterGraph method of the ICaptureGraphBuilder2 interface:

```
hResult = m_pCaptureGraphBuilder->SetFil
```

The SetFilterGraph method takes a pointer to the IGraphBuilder interface. This specifies which filter graph the capture graph builder will use. If you don't call the SetFilterGraph method, the Capture graph builder automatically creates a graph when it needs it.

Now, you're ready to create an instance of the video capture filter. The following code initializes a Video capture filter, the pointer of which is returned by the CoCreateInstance:

```
IBaseFilter *pVideoCaptureFilter

hResult=CoCreateInstance(CLSID_VideoCapt

                         IID_IBaseFilter
```

You then need to get a pointer to IPersistPropertyBag from the video capture filter. You use this pointer to set the capture device (in other words, the camera) that the capture filter will use, as follows:

```
IPersistPropertyBag *pPropertyBag;

hResult=pVideoCaptureFilter->QueryInterf
```

Now, you need to get a handle on the camera you will use to capture video. You can enumerate the available camera devices by using the FindFirstDevice and FindNextDevice functions. You can have multiple cameras present on a device. (HTC Universal is one example.) To keep the code simple for this example, use FindFirstDevice to get the first available camera on the device as follows:

```
DEVMGR_DEVICE_INFORMATION devInfo;

CComVariant  CamName;

CPropertyBag PropBag;

GUID guidCamera = { 0xCB998A05, 0x122C,

                  0x3E, 0x4D, 0x7E, 0x

devInfo.dwSize = sizeof(devInfo);

FindFirstDevice( DeviceSearchByGuid, &gu

CamName=devInfo.szLegacyName

PropBag.Write( _T("VCapName"), &CamName

pPropertyBag->Load( &PropBag, NULL );

hResult =pFilterGraph->AddFilter( pVideo

                                 _T("Vi
```


Page 6 of 17

```
pPropertyBag.Release();
```

Note the first parameter in the FindFirstDevice, DeviceSearchByGuid. It specifies the search type. Other options are DeviceSearchByLegacyName, DeviceSearchByDeviceName, and so forth. DeviceSearchByGuid is the most reliable way to find a capture device. The information regarding the device is returned in the DEVMGR_DEVICE_INFORMATION structure. You store the szLegacyName value in the CComVariant variable, and you need an object that has implemented IPropertyBag interface.

In the code sample, CPropertyBag is a custom class that has implemented IPropertyBag. This object is needed to pass the capture device name to the filter. The string VCapName identifies the filter property for the name of the video capture device. Once you have set the capture device, you can add the Video capture filter to the filter graph. You use the AddFilter method of the graph manager for this. This method takes two parameters: the first is the pointer to the filter that is to be added, and the second is the name of the filter. The second parameter can be NULL; in this case, the filter graph manager generates a unique name for the filter. If you have provided a name that conflicts with some other filter, the manager will modify the name to make it unique.

You then need to instantiate the WMV9 encoder:

```
IBaseFilter *pVideoEncoder;

IDMOWrapperFilter *pWrapperFilter;

hResult=CoCreateInstance(CLSID_DMOWrappe

                         IID_IBaseFilter

hResult =pVideoEncoder->QueryInterface( 

hResult =pWrapperFilter->Init( CLSID_CWM

                               DMOCATEGO

hResult=pFilterGraph->AddFilter( pVideoE
```

Because the WMV9 encoder is a DMO, you can't add/use it like other filters. But DirectShow provides a wrapper filter that enables you to use a DMO like any other filter. You first create an instance of the DMO wrapper filter and then initialize the WMV9 encoder DMO with it. After initializing the DMO, you add it into the filter graph as follows:

```
IBaseFilter *pASFMultiplexer;

IFileSinkFilter *pFileSinkFilter;

hResult = pCaptureGraphBuilder->SetOutpu
```

```
   &MEDIASUBTYPE_Asf, T("test.asf"), &pA

   &pFileSinkFilter );
```

You have added the source and the transform filter in the filter graph, so the last thing remaining is adding a sink filter in the graph. For this, you call the SetOutputFileName method of ICaptureGraphBuilder2. The first parameter is a media subtype; the second parameter is the name of the file in which you want to save the video; the third parameter is the address of a pointer that receives the multiplexer's interface; and the fourth parameter receives the file writers' interface.

With that, your filter graph is ready. All you need to do is connect the source filter, encoder, and multiplexer. You can achieve this by using the RenderStream method of the graph builder, as follows:

```
hResult = pCaptureGraphBuilder->RenderSt
```

The first parameter is the pin category, which can be NULL to match any category.

The second parameter specifies the media type. The third, fourth, and fifth parameters specify a starting filter, an intermediate filter, and a sink filter, respectively. The method connects the source filter to the transform filter and then the transform filter to the sink filter.

Now your graph is ready, and you can start capturing the video.

Controlling the Graph

Before capturing video, you need two more things: the ImediaEventEx and IMediaControl pointers. IMediaEventEx derives from IMediaEvent, which supports event notification from the filter graph and individual filters to the application. ImediaEventEx provides a method to the register window that receives a message when any event occurs.

IMediaControl is an interface exposed by the filter graph that allows an application to control the streaming media through the graph. The application can use this to start, stop, or pause the running graph.

The following code sample first queries the filter graph for its IMediaEventEx interface. Once it gets the pointer to the IMediaEventEx interface, it then calls its method SetNotifyWindow, passing it the handle to the window that handles the message. The second parameter is the message that will be passed as notification to the Windows message handler. The third parameter is the instance data (this can be 0):

```
IMediaEventEx *pMediaEvent;

IMediaControl *pMediaControl;

#define WM_GRAPHNOTIFY WM_APP+1

hResult =pFilterGraph->QueryInterface( I

                                      &

hResult =pMediaEvent->SetNotifyWindow((O

                                     WM

hResult=pFilterGraph->QueryInterface(&pM

hResult =pMediaControl->Run();
```

When an event occurs, DirectShow will send WM_GRAPHNOTIFY to the specified windows.

Note: *WM_GRAPHNOTIFY is used here as an example. This can be any application-defined message.*

Next, you get the pointer to the IMediaControl interface. You'll use this interface to control the graph. Call its Run method to put the entire graph into a running state. The following code shows how to start and stop capture by throwing the ControlStream method of CaptureGraphBuilder:

```
LONGLONG dwStart  = 0, dwEnd      = 0;

WORD wStartCookie = 1, wEndCookie = 2;

dwEnd=MAXLONGLONG

//start capturing

hResult=pCaptureGraphBuilder->ControlStr

   &PIN_CATEGORY_CAPTURE, &MEDIATYPE_Vid

   pVideoCaptureFilter, &dwStart, &dwEnd

   wStartCookie, wEndCookie );

//Stop capturing

dwStart=0;

hResult=pFilterGraph->QueryInterface(&pM
```

```
hResult=pMediaSeeking->GetCurrentPositio

hResult= pCaptureGraphBuilder->ControlSt

   &PIN_CATEGORY_CAPTURE, &MEDIATYPE_Vid

   &dwStart, &dwEnd, wStartCookie, wEndC
```

The code uses the search criteria supplied in the method call to locate an output pin on the capture filter. ControlStream enables an application to control streams without it needing to enumerate filters and pins in the graph.

Start and End specify the start and stop times (MAX_LONGLONG is the largest possible reference time value). When you start, the End is set to MAXLONLONG. When you want to stop, you first get the current position of the stream by using the GetCurrentPosition method of the IMediaSeeking interface. You then call the ControlStream method with Start set at 0 and End set at the current position.

You now have the graph ready and running. You can start using it to capture and save in an .asf file.

Handling the Graph Events

Because an application will control the graph, you need to write the code to facilitate that. You already have registered the window and message with the filter graph, so the only thing remaining is to handle the message in the window's message handler as follows:

```
BOOL CALLBACK VidCapDlgProc(HWND hDlg,UI
                            LPARAM lPara
{
   … … … …
   case WM_GRAPHNOTIFY:
      {
         ProcessGraphMessage();
      }
   … … … …
}
ProcessGraphMessage()
{
   HRESULT hResult=S_OK;
   long leventCode, param1, param2;
```


Page 14 of 17

```
while(hResult=pEvent->GetEvent(&leven
SUCCEEDED(hResult))
{
   hResult = pEvent->FreeEventParams(
   if (EC_STREAM_CONTROL_STOPPED == l
   {
      pMediaControl->Stop();
      break;
   }
   else if(EC_CAP_FILE_COMPLETED== le
   {
      //Handle the file capture compl
   }
   else if(EC_CAP_FILE_WRITE_ERROR==
   {
      //Handle the file write error e
   }
```


Page 15 of 17

```
    }

}
```

You handle the WM_GRAPHNOTIFY message in the windows handler. DirectShow sends this message to the application when any event arises. The application calls a user-defined method to process the events. The GetEvent method of the IMediaEvent interface retrieves the event code and two event parameters from the queue.

Because the message loop and event notification are asynchronous, the queue might hold more then one event. Hence, the GetEvent code is called in a loop until it returns a failure code. Also, whenever you call GetEvent, it's important to call FreeEvent to free the resource associated with the event parameter. And, being the good programmer that you are, you won't forget to release the resources afterwards, will you? Call Release on every object that you have created, as follows:

```
PVideoCaptureFilter->Release ();

pVideoEncoder->Release ();

pMediaEvent ->Release();

pMediaSeeking ->Release();
```


Page 16 of 17

```
pASFMultiplexer->Release();

pFileSinkFilter->Release();

pWrapperFilter ->Release();

pFilterGraph->Release();

pCaptureGraphBuilder->Release();
```

What Have You Learned?

You now understand how to create, run, and control a filter graph manually. By using the DirectShow framework to capture from a camera, you gain good control with ease.

About the Author

Amit Ranjan, a Windows Embedded MVP who speaks at various Microsoft conferences and user group sessions, has six years of professional experience. He currently works with Sasken Communication Tech Ltd. in Bangalore, India.