Chapter 25

DirectShow Capture

Why Read This Chapter?

As we were writing the book, DirectShow was still in its beta cycle. Nonetheless, we decided to add a new chapter to cover some of the new features of DirectShow, namely the new capture and compression interfaces.

In this chapter, you'll

- get an overview of capture under VFW and DirectShow;
- understand the internal workings of the sample video capture filter, which wraps any VFW driver; and
- learn how to access the new capture interfaces from your application.

25.1 Overview of DirectShow Capture

As of the time of the publication of this book, all video capture adapters use the Video for Windows (VFW) interface to implement capture drivers under Windows. VFW was designed to handle linear video capture and compression but did not handle video/audio synchronization, MPEG-style video, TV tuning, or video conferencing. As a result, Microsoft developed ActiveMovie 1.0 to address some of these issues. ActiveMovie 1.0 provided audio and video playback, synchronization between multiple streams, and MPEG1 support. But ActiveMovie 1.0 lacked support for audio/video capture, TV tuning, and compression.

DirectShow 2.0 completes the puzzle and provides the missing pieces of ActiveMovie 1.0. DirectShow defines the necessary interface to build a capture source filter and the necessary means of accessing such interfaces from within an application. To provide a migration path, Microsoft implemented a default capture filter, which acts as a wrapper for any existing VFW video capture filter. A sample filter that demonstrates the VFW wrapper is provided in the DirectShow SDK. We'll be using this sample filter for our analysis of DirectShow's capture/compression interfaces.

25.1.1 Issues with a VFW Capture Filter

Even though the default VFW capture filter, supplied with the DirectShow SDK, provides an instantaneous port to any VFW capture driver, it has a couple of issues that could negatively affect capture performance:

- the capture filter must perform a memory copy of the data from a VFW buffer to a DirectShow buffer; and
- under Windows 95, since VFW is not completely implemented in 32 bits, VFW continuously switches between the 16- and 32-bit operating system modes, a.k.a thunking, which has a negative impact on performance.

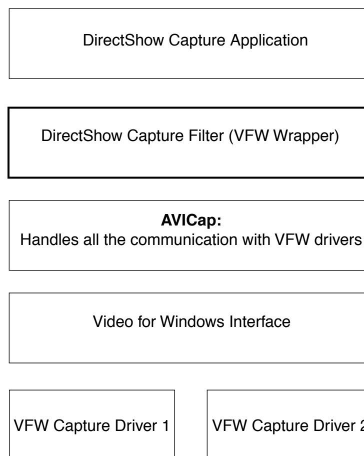


Figure 25-1 DirectShow capture filter wrapper for existing VFW capture drivers.

Let's explain. The default capture filter uses the AVICap interface to communicate with the VFW capture driver. Microsoft defined the AVICap interface to ease the task of developing video capture applications under VFW. AVICap handles all the buffer management, callbacks, and communication with the VFW capture driver. It also handles writing the captured data into a file and pre-allocates the output file if requested by the user.

Why is it a problem? AVICap allocates the video capture buffers and passes them to the VFW driver. When the data is ready, the VFW driver writes the captured image into one of those buffers and makes a callback to AVICap, which in turn calls the DirectShow capture filter with that video buffer—remember AVICap allocated that buffer. Now, since the DirectShow capture filter negotiates a shared buffer with the downstream filter, it must copy the data from AVICap's buffer to the shared buffer and deliver it to the downstream filter. The extra memory required to copy the data between buffers has a negative impact on performance.

So what's the issue with thunking? Even though Windows 95 is a 32-bit operating system, it still contains many 16-bit components. For example, the Video for Windows capture interface and drivers remain as 16-bit entities. In order for a 32-bit application to access the capture driver, Microsoft provided a 32-bit VFW wrapper that switches back and forth between the 32-bit application and the 16-bit VFW capture drivers. All of this thunking is very expensive and degrades capture performance. As you recall, the default 32-bit DirectShow capture filter uses AVICap to wrap any 16-bit VFW driver; therefore it suffers the same thunking penalties as any 32-bit VFW application.

Keeping this in mind, we think that the default DirectShow capture filter (wrapper) is an excellent migration point for existing VFW capture drivers. But, depending on the performance of your VFW capture filter with this video wrapper, you might want to port your VFW capture driver into a native 32-bit DirectShow capture filter. But before you start coding, you might want to wait for the Windows Driver Model (WDM) capture interface to be fully defined and functional—it is coming soon in Windows 98.

On the application front, you don't have to wait for WDM capture to be fully defined. You can immediately use the new DirectShow capture interface to access existing VFW capture filters, and, in the future, you can access the next generation of WDM capture filters without modifying your application.

25.1.2 The WDM Capture Model

Similar to VFW, DirectShow provides a capture architecture that runs at the Windows user level—non-kernel. This model results in a high-latency response for the capture drivers, which could result in delays in delivering the data. For instance, a video conferencing application dictates a latency bound from the time the video/audio is captured by the capture adapter to the time when the video frame is delivered and displayed on the receiving

end. WDM offers a streaming model that runs at kernel privilege level, and thus eliminates such latency. (See Figure 25-2.)

In addition, the current VFW architecture lacks some essential feature for video conferencing, TV viewing with multiple fields, and data streaming in a vertical blanking interval (VBI) of a TV signal. Such features require tight coupling between the video capture driver and the hardware—which is not easily provided with the VFW architecture. The WDM video capture model provides support for USB conferencing cameras, 1394 DV devices, desktop cameras, TV viewing, multiple video streams support, and VPE capture support. This support is provided through kernel-based streaming.

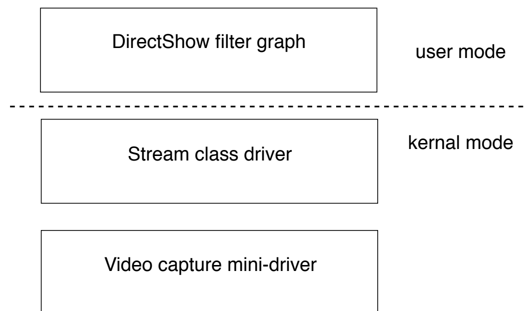


Figure 25-2 DirectShow and WDM streaming models.

WDM provides a kernel driver model that allows you to build a "WDM filter," a.k.a. a mini-driver, which allows you to stream data at the kernel level. The WDM model can easily tie into the DirectShow filter graph and exchange data with a user-level DirectShow filter. Previously, DirectShow filters transmitted data to and from the kernel whenever necessary to decompress a video sequence or to render the final frame. However, each of these transitions is time-consuming because the transition takes time, and because of the parameter validation, security validation, and possibly data copying that must occur.

Through kernel streaming, a stream makes fewer transitions between the user and the kernel mode. It can be either partially or entirely produced and consumed in kernel mode. When streams are processed in kernel mode, the DirectShow filters merely expose control mechanisms to direct the streams in the kernel mode—they do not touch the data. This greatly reduces the overhead associated with numerous transitions between modes.

Kernel streams can pass data to the filter graph at appropriate points, depending on the application. Figure 25-3 illustrates the transition of data from a DirectShow filter (user mode) to a WDM driver (kernel mode). The data is entirely consumed at the kernel level, where it is decoded, color converted, and rendered to the screen. The skeleton DirectShow Filter graph (proxies) are created in order to control the WDM stream and to communicate with the application.

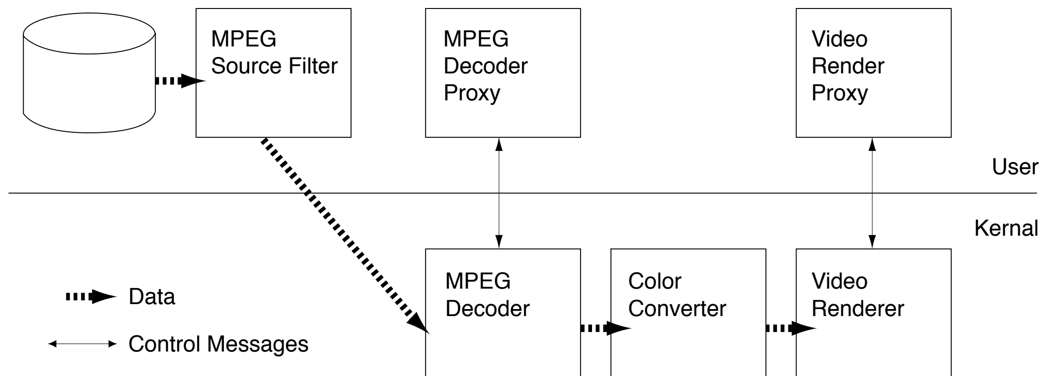


Figure 25-3 An example of WDM and DirectShow streaming.

25.2 A Look at the Capture Filter Graph

As we mentioned earlier, the DirectShow SDK provides a sample capture filter, which shows how to wrap any existing VFW drivers. Although the code is very detailed and involves lots of code, it should be easy to follow with the aid of this chapter. Rather than duplicating the effort and writing our own capture filter, we chose to use this sample filter to demonstrate how to write a video capture filter. We decided to focus our attention on the application capture interface since there are more application developers out there than the handful of capture adapter vendors.

Let's start by looking at an audio/video capture filter graph (Figure 25-4). The diagram shows the following filters:

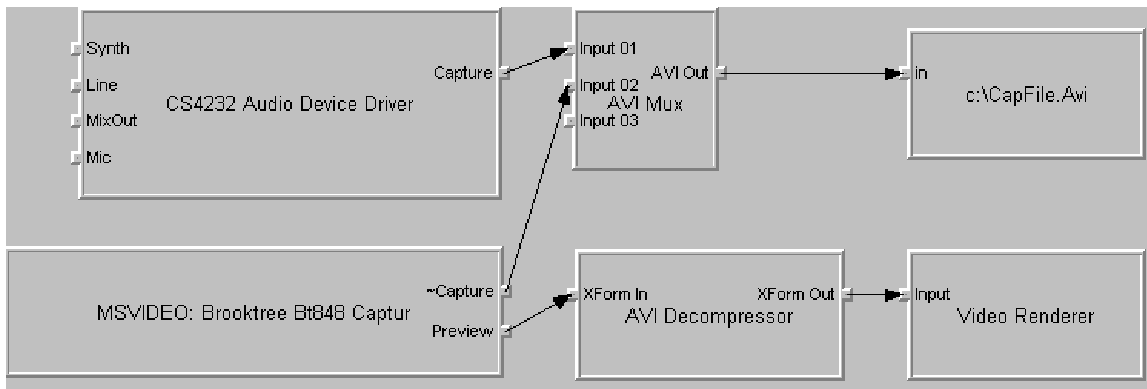


Figure 25-4 Video and audio capture filter graph with video preview.

CS4232 Audio Device Driver. This is an audio capture source filter that captures audio data from a sound adapter. Notice that, unlike the file source filters discussed in the DirectShow filters chapter, the audio capture source filter includes four input pins: synth, line, mixout and mic. In general, capture source filters, both audio and video, are allowed to have input pins in order to select the source of data and where it is coming from. With audio capture filters, you can select between a synthesizer, a line, a mixer, or a microphone as an input device. The digitized data is sent out to the output pin capture.

MSVideo: Brooktree Bt848 Capture. This video capture filter is basically a wrapper to existing VFW capture driver filters. It is supplied as part of the DirectShow SDK and allows any existing VFW driver to be used without modification—unless, of course, there are proprietary interfaces in the VFW driver. The video capture driver includes two pins: *capture* and *preview.* The capture pin provides the real-time captured data and sends it out to the downstream filter. The preview pin allows the user to view what's being recorded. Notice that a capture filter is required to have only the capture pin; however, if it had the preview pin, both pins would have to output the same data type (YUV9, RGB8, and so forth).

AVI Decompressor and Video Renderer. Since our capture driver was set to generate YUV9 data, the output of the preview pin had to be color converted to RGB and displayed by the video renderer.

AVI Mux. This filter mixes multiple audio and video streams into the Audio Video Interleaved (AVI) format and sends the output to the downstream filter.

File Writer Filter. This filter takes the AVI data and writes it out to the specified output file, *C:\capfile.avi*. This filter has no knowledge of the data; it just writes the data to the file.

25.3 Writing a Video Capture Filter

As we mentioned earlier, a video capture filter is basically a source filter that reads its data from a video capture adapter, a TV tuner, or a USB digital camera. Just like any source filter, a video capture filter is based on the CSource interface, and the output capture pin is based on the CSourceStream interface. Yet to expose the functionality required by video capture applications, a video capture filter implements additional interfaces, which allows applications to control the capture filter, as shown in Figure 25-5. We'll discuss these interfaces in more detail later in this chapter (see section 25.5).

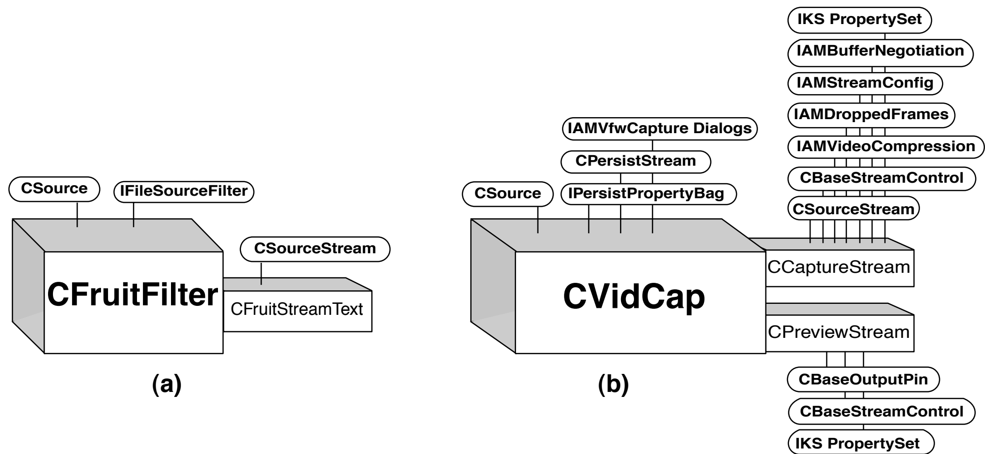


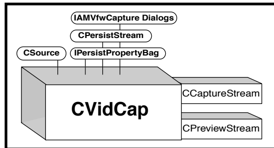
Figure 25-5 (a) file source filter, (b) video capture source filter.

In order for a source filter to qualify as a capture filter, it must have an output capture pin, which supports the IKSPropertySet interface. The capture pin must also declare itself of the type `PIN_CATEGORY_CAPTURE`. The preview pin is optional on the capture filter, but if it exists, it must also support the IKSPropertySet interface and must declare itself of the type `PIN_CATEGORY_PREVIEW`. We'll see how to support this later in this chapter.

In the following analysis, we'll use the sample capture filter provided in the DirectShow SDK (the VidCap sample). We'll first look at the body of the capture source filter CVidCap, then we'll examine the capture pin CCapture-Stream followed by the preview pin CPreviewStream. We'll discuss each of

the new interfaces and point out which functions you need to override and why.

25.4 The Body of The Capture Source Filter: CVidCap



As with any DirectShow source filter, a video capture filter is based on the CSource interface, which handles the enumeration of pins exposed by the source filter. However, unlike the default implementation of CSource, which assumes that all the output pins are derived from the CSourceStream interface, the capture filter derives the capture pin from CSourceStream and the preview pin from the CBaseOutputPin interface. By doing this, the capture filter has more control over the creation and flow of data across its output pins.

Why? The capture filter has to figure out the level of functionality the capture adapter supports before it creates the output pins. For example, if the capture adapter has overlay support, the capture filter creates a hardware overlay pin; otherwise it creates a software preview pin. Both overlay and preview pins allow the user to view the video as it is being captured. To achieve this level of control, the capture filter overrides the pin negotiation functions of the base CSource interface and responds to them directly.

Let's first look at the declaration of the CVidCap filter. Notice that the filter is based on the CSource interface. It is also based on the IPersistProperty-Bag interface, which allows the filter to load and save its configuration information in the system registry. CVidCap is also based on the CPersist-Stream interface, which allows it to load and save its configuration to the filter graph file (*.grf*) file (*.grf* files are generated by the Graph Editor application that ships with the DirectShow SDK). Finally, the filter is also based on the IAMVFWCaptureDialogs interface, which allows it to display the VFW driver dialog boxes. Notice that this interface is provided only for filters that can wrap VFW drivers and is not normally required by Direct-Show's capture filters.

```
class CVidCap :
   public CSource, // base source filter
   public IPersistPropertyBag,// handles configuration information in registry
   public CPersistStream// handles configuration information in GRF files
   public IAMVFWCaptureDialogs, // handles VFW dialog boxes
{
public:
   DECLARE_IUNKNOWN
   STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void ** ppv);
```

```
    // Construct a VidCap filter
    static CUnknown * WINAPI CreateInstance(LPUNKNOWN lpunk, HRESULT *phr);
    CCritSec m_cStateLock;// Lock this when a function accesses the filter state. Generally _all_
      // functions, since access to this filter will be by multiple threads.

    // CSource functions
    int GetPinCount();
    CBasePin * GetPin(int ix);

    // overide the Run & Pause methods so that I can notify the pin
    // of the state it should move to - also needed for CBaseStreamControl
    STDMETHODIMP Run(REFERENCE_TIME tStart);
    STDMETHODIMP Pause(void);
    STDMETHODIMP Stop(void);

    // live source - override GetState to return VFW_S_CANT_CUE when pausing
    // since we won't be sending any data when paused
    STDMETHODIMP GetState(DWORD dwMSecs, FILTER_STATE *State);

    // IAMVFWCaptureDialogs stuff
    STDMETHODIMP HasDialog(int iDialog);
    STDMETHODIMP ShowDialog(int iDialog, HWND hwnd);
    STDMETHODIMP SendDriverMessage(int iDialog, int uMsg, long dw1, long dw2);

    // for IAMStreamControl
    STDMETHODIMP SetSyncSource(IReferenceClock *pClock);
    STDMETHODIMP JoinFilterGraph(IFilterGraph * pGraph, LPCWSTR pName);

    // IPersistPropertyBag methods
    STDMETHODIMP InitNew();
    STDMETHODIMP Load(LPPROPERTYBAG pPropBag, LPERRORLOG pErrorLog);
    STDMETHODIMP Save(LPPROPERTYBAG pPropBag, BOOL fClearDirty, BOOL fSaveAllProperties);

    // CPersistStream
    STDMETHODIMP GetClassID(CLSID *pClsid);
    int SizeMax();
    HRESULT WriteToStream(IStream *pStream);
    HRESULT ReadFromStream(IStream *pStream);

    STDMETHODIMP FindPin(LPCWSTR Id, IPin ** ppPin);
};
```

Since the number of pins and their base class could differ depending on the functionality of the capture adapter, CVidCap overrides both the *CSource::GetPinCount()* and *CSource::GetPin()* functions to expose its pins accordingly. The *GetPinCount()* function returns the number of pins created for this filter. The *GetPin()* function returns a pointer to each of the pins, namely capture, preview, and overlay.

```
int CVidCap::GetPinCount()
{
  // we have a preview pin (one or the other)
  if (m_pOverlayPin || m_pPreviewPin)
                              return 2;
  else if (m_pCapturePin)
                              return 1;
```

```
  else
                                return 0;
}

CBasePin * CVidCap::GetPin(int ii)
{
  if (ii == 0 && m_pCapturePin) // capture pin is first one in list
      return m_pCapturePin;
  if (ii == 1 && m_pOverlayPin) // Overlay or Preview is second pin - either or
      return m_pOverlayPin;
  if (ii == 1 && m_pPreviewPin)
      return m_pPreviewPin;
  return NULL;
}
```

The capture filter also overrides the function *CSource::SetSyncSource()* to allow applications to set the source of the synchronization clock of audio/video streams. This function passes the new clock source to the output pins—they're the ones that handle all the synchronization and delivery of data.

```
STDMETHODIMP CVidCap::SetSyncSource(IReferenceClock *pClock)
{
   if (m_pCapturePin)
   m_pCapturePin->SetSyncSource(pClock);
   if (m_pPreviewPin)
   m_pPreviewPin->SetSyncSource(pClock);
   return CSource::SetSyncSource(pClock);
}
```

The capture filter also overrides the *CSource::JoinFilterGraph()* function, which provides a pointer to the filter graph, pGraph. The capture filter calls the base function *CSource::JoinFilterGraph(),* which saves the address of the filter graph and queries the filter graph for its IMediaEvenSink interface. The capture filter passes the IMediaEventSink pointer to its output pin, which uses the pointer to send notification messages to the filter graph manager or to the application.

```
STDMETHODIMP CVidCap::JoinFilterGraph(IFilterGraph * pGraph, LPCWSTR pName)
{
    HRESULT hr = CSource::JoinFilterGraph(pGraph, pName);
    if (hr == S_OK && m_pCapturePin)
      m_pCapturePin->SetFilterGraph(m_pSink);
    if (hr == S_OK && m_pPreviewPin)
      m_pPreviewPin->SetFilterGraph(m_pSink);
    return hr;
}
```

25.4.1 Saving and Loading Capture Device Configurations: IPersistPropertyBag

Since a capture filter can support multiple capture devices, the filter needs to know when to load/save the appropriate configuration of the capture device. The capture filter implements the IPersistPropertyBag interface in order to handle the loading/storing of the capture device configuration. In VFW, the information about capture devices is kept in the [drivers32] section of the *system.ini* file. For example, the `msvideo` and `msvideo1` entries, shown below, specify two capture adapters, ISRV3.Drv and XYZCap.Drv. The VFW application looks at these values in the *system.ini* file and decides to load the appropriate VFW driver for the capture adapter using VFW functions.

```
System.Ini:

[drivers32]              ; This shows two capture adapters ISVR3 and XYZCap
msvideo=ISVR3.Drv
msvideo1=XYZCap.drv

[ISVR3.Drv]              ; configuration for ISVR3
IRQ=10
IO=320
Overlay=0

[XYZCap.Drv]             ; Configuration for XYZCap
IRQ=5
IO=3a0
```

In DirectShow, the list of capture devices and their configuration are saved in the system registry. A DirectShow filter implements the IPersistPropertyBag COM interface in order to load/store configuration information from the registry. The filter overrides a set of the IPersistPropertyBag interface functions, which are called when the filter is loaded or unloaded. They give the filter the chance to read or save its configuration information to a storage unit—the registry in our case.

The sample capture filter implements the IPersistPropertyBag functions: *Load()* , *Save()*, and *InitNew()*. The *InitNew()* function is called to initialize a new instance of the filter's property bag. The *Load()* function is then called to allow the filter to read its configuration information out of the registry and load the appropriate capture device. Remember, there might be more than one capture adapter in the system. The *Load()* function is equivalent to the `DRV_LOAD` message in VFW. The *Save()* function is called to

allow the filter to save any configuration information that could have changed during the life of the filter; this information could be used the next time around when the filter is loaded. Notice that both functions accept a pointer to a PROPERTYBAG object, which points to an entry in the system registry. We'll discuss this topic in section 25.6.1.

```
STDMETHODIMP CVidCap::InitNew()
{
  // if we already created a capture pin, then we're alrady initialized
  if(m_pCapturePin)
     return HRESULT_FROM_WIN32(ERROR_ALREADY_INITIALIZED);

  return S_OK;
}

STDMETHODIMP CVidCap::Load(LPPROPERTYBAG pPropBag, LPERRORLOG pErrorLog)
{
  HRESULT hr = S_OK;
  CAutoLock l(pStateLock());

  // No property bag => Default to capture device #0
  // Note: CreatePins() uses the m_iVideoId to figure out which device to
load
  if (pPropBag == NULL) {
     m_iVideoId = 0;
     CreatePins(&hr);
     return hr;
    }

  // find out what device to use.  Read it from the registry
  // Note: CreatePins() uses the m_iVideoId to figure out which device to
load
  VARIANT var;
  var.vt = VT_I4;
  hr = pPropBag->Read(L"AviCapIndex", &var, 0);
  m_iVideoId = var.lVal;
  CreatePins(&hr);
  return hr;
}

STDMETHODIMP CVidCap::Save(
  LPPROPERTYBAG pPropBag, BOOL fClearDirty, BOOL fSaveAllProperties)
{
  // nothing to save, but still must return S_OK.
  return S_OK;
}
```

25.4.2 Saving and Loading *.grf* Files: CPersistStream

In addition to saving configuration information into the registry, a capture filter, or any filter indeed, is required to implement the CPersistStream if it wants to save information in a *.grf* file. As we mentioned in Chapter 9, a *.grf* file allows you to save an entire filter graph with all the connection information.

Typically, when the *.grf* file is saved, the function *GetClassID()* is called to retrieve the class ID of the filter and save it in the *.grf* file. The function *SizMax()* is then called to determine how many bytes of data to allocate for this filter in the *.grf* file. Finally, the *WriteToStream()* function is called to write any information that the filter wants to save—in our case it saves the device ID.

When the *.grf* file is loaded and rendered, the filter graph manager determines which filter to load from the class ID that it saved in the *.grf* file. The appropriate filter is loaded into the filter graph, and the function *ReadFromStream()* is then called in order to read its information out of the *.grf* file. In our case, we read the device ID so we can re-create the filter graph for that device.

```
STDMETHODIMP CVidCap::GetClassID(CLSID *pClsid)
{
    CheckPointer(pClsid, E_POINTER);
    *pClsid = CLSID_VidCap;
    return S_OK;
}

int CVidCap::SizeMax()
{
    // We only need a long int (m_iVideoId)
    return sizeof(m_iVideoId);
}

HRESULT CVidCap::WriteToStream(IStream *pStream)
{
    // We only care about the device ID, let's save that for next time.
    return pStream->Write(&m_iVideoId, sizeof(m_iVideoId), 0);
}

HRESULT CVidCap::ReadFromStream(IStream *pStream)
{
    HRESULT hr = S_OK;

    // Read the Device ID from the GRF file and create the appropriate pins
    hr = pStream->Read(&m_iVideoId, sizeof(m_iVideoId), 0);
    CreatePins(&hr);
    return hr;
}
```

Notice that the capture filter also overrides the *CSource::FindPin()* function, which is also used for consistency. In a typical implementation of a source filter, the default implementation of the CSource class handles the *FindPin()* function. But, since the capture filter supports output pins that are not based on the CSourceStream interface, we override this function and call the base *CBaseOutputPin::FindPin()* function instead to provide a

matching implementation for all the output pins—including the capture pin.

```
STDMETHODIMP CVidCap::FindPin(LPCWSTR Id, IPin ** ppPin)
{
    return CBaseFilter::FindPin(Id, ppPin);
}
```

25.4.3 Support for VFW Dialog Boxes: IAMVFWCaptureDialog

The DirectShow capture implementation includes the interface IAMVFW-CaptureDialog, which enables applications to access the configuration dialog boxes exposed by VFW capture drivers. Typically VFW drivers display configuration dialog boxes, which allow the user to change the resolution, color format, or the input source. DirectShow provides the IAMVFWCaptureDialog to allow applications to display these same dialog boxes.

The *IAMVFWCaptureDialog::HasDialog()* function declares which dialog boxes are supported by this filter. In our case we return the level of support of the VFW driver. Typically, a VFW driver can display up to three dialog boxes: a Video Source dialog box, which allows the user to select the source of video capture (S-Video or Composite); a Video Format dialog box, which allows the user to select the capture format and image size; and a Video Display dialog box, which allows the user to select between overlay and preview.

```
HRESULT CVidCap::HasDialog(int iDialog)
{
    if (iDialog == VFWCaptureDialog_Source)
      return (m_pCapturePin->m_SupportsVideoSourceDialog ? S_OK :
S_FALSE);
    else if (iDialog == VFWCaptureDialog_Format)
      return (m_pCapturePin->m_SupportsVideoFormatDialog ? S_OK :
S_FALSE);
    else if (iDialog == VFWCaptureDialog_Display)
      return (m_pCapturePin->m_SupportsVideoDisplayDialog ? S_OK :
S_FALSE);

    return E_INVALIDARG;
}
```

Once the application figures out which dialogs are supported, it can call the *IAMVFWCaptureDialog::ShowDialog()* function to display those dialogs.

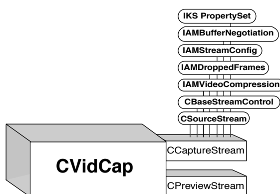
```
HRESULT CVidCap::ShowDialog(int iDialog, HWND hwnd)
{
    HRESULT hr;

    // we can't hold any critical sections while the dialog box is up before bringing up a dialog
that
    // could change our format, make sure we're not streaming
    if (m_State != State_Stopped)
     return E_UNEXPECTED;

    if (iDialog == VFWCaptureDialog_Source)
        capDlgVideoSource(m_pCapturePin->m_hwCapCapturing);
    else if (iDialog == VFWCaptureDialog_Format)
        capDlgVideoFormat(m_pCapturePin->m_hwCapCapturing);
    else if (iDialog == VFWCaptureDialog_Display)
        capDlgVideoDisplay(m_pCapturePin->m_hwCapCapturing);

    // bringing up the Format Dialog can change the format we are capturing with.
    if (iDialog == VFWCaptureDialog_Format) {
        // See the SDK sample source for details …
         :  :  :
     }
}
```

25.5 The Capture Pin: CCaptureStream



As you can see in the class declaration below, the capture pin supports a huge list of interfaces in order to become a capture pin. The capture pin supports the CSourceStream interface, which provides the standard media type negotiation, buffer management, data delivery, and thread handling functions. As we mentioned earlier, the capture pin must support the IKsPropertySet interface in order to declare itself as a capture pin. The pin also supports the *CBaseStreamControl, IAMVideoCompression*, *IAM-StreamConfig, IAMDroppedFrames*, and *IAMBufferNegotiation* functions. We'll discuss these functions below.

```
class CCaptureStream :
     public CSourceStream,
     public CBaseStreamControl,
     public IAMVideoCompression,
     public IAMStreamConfig,
     public IAMDroppedFrames,
     public IAMBufferNegotiation,
     public IKsPropertySet
{
public:
     //  --- CSourceStream implementation ---
    HRESULT DecideBufferSize(IMemAllocator *pAlloc, ALLOCATOR_PROPERTIES *pProperties);
```

```
    // IKsPropertySet stuff - to tell the world we are a "capture" type pin
    STDMETHODIMP Set(REFGUID guidPropSet, DWORD dwPropID, LPVOID pInstanceData,
        DWORD cbInstanceData, LPVOID pPropData, DWORD cbPropData);
    STDMETHODIMP Get(REFGUID guidPropSet, DWORD dwPropID, LPVOID pInstanceData,
        DWORD cbInstanceData, LPVOID pPropData, DWORD cbPropData,
        DWORD *pcbReturned);
    STDMETHODIMP QuerySupported(REFGUID guidPropSet, DWORD dwPropID,
        DWORD *pTypeSupport);

    // IAMStreamConfig stuff
    STDMETHODIMP SetFormat(AM_MEDIA_TYPE *pmt);
    STDMETHODIMP GetFormat(AM_MEDIA_TYPE **ppmt);
    STDMETHODIMP GetNumberOfCapabilities(int *piCount, int *piSize);
    STDMETHODIMP GetStreamCaps(int i, AM_MEDIA_TYPE **ppmt, LPBYTE pSCC);

    /* IAMVideoCompression methods */
    STDMETHODIMP GetInfo(LPWSTR pstrVersion, int *pcbVersion, LPWSTR pstrDescription,
            int *pcbDescription, long *pDefaultKeyFrameRate, long *pDefaultPFramesPerKey,
            double *pDefaultQuality, long *pCapabilities);

    STDMETHODIMP put_KeyFrameRate(long KeyFrameRate) {return E_NOTIMPL;};
    STDMETHODIMP get_KeyFrameRate(long *pKeyFrameRate) {return E_NOTIMPL;};
    STDMETHODIMP put_WindowSize(DWORDLONG WindowSize) {return E_NOTIMPL;};
    STDMETHODIMP get_WindowSize(DWORDLONG *pWindowSize) {return E_NOTIMPL;};
    STDMETHODIMP put_PFramesPerKeyFrame(long PFramesPerKeyFrame) {return E_NOTIMPL;};
    STDMETHODIMP get_PFramesPerKeyFrame(long *pPFramesPerKeyFrame) {return E_NOTIMPL;};
    STDMETHODIMP put_Quality(double Quality) {return E_NOTIMPL;};
    STDMETHODIMP get_Quality(double *pQuality) {return E_NOTIMPL;};
    STDMETHODIMP OverrideKeyFrame(long FrameNumber) {return E_NOTIMPL;};
    STDMETHODIMP OverrideFrameSize(long FrameNumber, long Size){return E_NOTIMPL;};

    /* IAMBufferNegotiation methods */
    STDMETHODIMP SuggestAllocatorProperties(const ALLOCATOR_PROPERTIES *pprop);
    STDMETHODIMP GetAllocatorProperties(ALLOCATOR_PROPERTIES *pprop);


    /* IAMDroppedFrames methods */
    STDMETHODIMP GetNumDropped(long *plDropped);
    STDMETHODIMP GetNumNotDropped(long *plNotDropped);
    STDMETHODIMP GetDroppedInfo(long lSize, long *plArray, long *plNumCopied);
    STDMETHODIMP GetAverageFrameSize(long *plAverageSize);

    // --- Worker Thread fn's ---
    HRESULT OnThreadCreate(void);
    HRESULT OnThreadDestroy(void);
    HRESULT DoBufferProcessingLoop(void);
    HRESULT FillBuffer(IMediaSample *pSamp);
    HRESULT Inactive(void);

    // Override to handle quality messages
    STDMETHODIMP Notify(IBaseFilter * pSender, Quality q);

    // Override to handle Persistency of GRF files
    STDMETHODIMP QueryId(LPWSTR * Id);

private:
    HRESULT GetMediaType(CMediaType *pmt);
    HRESULT CheckMediaType(const CMediaType *pmt);
    HRESULT SetMediaType(const CMediaType *pmt);
    HRESULT BreakConnect();
};
```

25.5.1 The IKsPropertySet Interface

This interface is necessary for both the capture and the preview pins. Applications use this interface to figure out what category the pin supports. In order to be considered a capture filter, the capture and preview pin must respond to the IKsPropertySet interface functions *QuerySupported(), Get(),* and *Set().*

Typically, an application calls the *QuerySupported()* function to figure out if the *Get()* or *Set()* functions are supported. The *QuerySupported()* function returns `S_OK` if the filter supports either the property *Get()* or *Set()* functions and if the combination of property set, `GUIDPROPSET`, and ID, `DWPROPID` are valid; otherwise it returns an error. In our case, we declare that we support the *Get()* function by setting `KSPROPRTY_SUPPORT_GET` in the third parameter.

```
HRESULT CCaptureStream::QuerySupported(REFGUID guidPropSet, DWORD dwPropID, DWORD
*pTypeSupport)
{
    if ( (guidPropSet != AMPROPSETID_Pin) || (dwPropID != AMPROPERTY_PIN_CATEGORY) )
        return E_PROP_ID_UNSUPPORTED;

    if (pTypeSupport)
        *pTypeSupport = KSPROPERTY_SUPPORT_GET;
    return S_OK;
}
```

The application then calls the *Get()* function to figure out what kind of a pin this is. The *Get()* function returns `PIN_CATEGORY_CAPTURE` in the pPropData buffer, declaring that it is a capture pin. This is how the application knows that this is a capture pin and then figures out how to connect this filter in the filter graph.

```
HRESULT CCaptureStream::Get(REFGUID guidPropSet, DWORD dwPropID, LPVOID pInstanceData,
DWORD cbInstanceData, LPVOID pPropData, DWORD cbPropData, DWORD *pcbReturned)
{
    // make sure that the Property set and the pin category are valid
    if ( (guidPropSet != AMPROPSETID_Pin) || (dwPropID != AMPROPERTY_PIN_CATEGORY) )
        return E_PROP_ID_UNSUPPORTED;

    // make sure that we have at least one pointer to write either the size or data in
    if (pPropData == NULL && pcbReturned == NULL)
        return E_POINTER;

    // indicate how many bytes we're filling in
    if (pcbReturned)
        *pcbReturned = sizeof(GUID);
```

```
    // If they're both NULL, the caller wanted the size of buffer to allocte (in pcbReturned)
    if ( (pPropData == NULL) && (cbPropData == 0) )
        return S_OK;

    // We have space to put the data in, declare that we are a capture pin.
    *(GUID *)pPropData = PIN_CATEGORY_CAPTURE;
    return S_OK;
}
```

25.5.2 The CSourceStream Interface

The CSourceStream interface is the same interface used for the source filter, which is the output pin for the captured data. In the capture filter, we'll override the same functions as we did in Chapter 8, such as *SetMediaType(), CheckMediaType(), GetMediaType(), OnThreadCreate(), OnThreadDestroy(), DecideBufferSize()* and *FillBuffer()*. In addition, we'll override the *DoBufferProcessing(), QueryID(), InActive(),* and *BreakConnect()* functions for the capture filter.

25.5.3 Which Format Does the Filter Support? IAMStreamConfig

The IAMStreamConfig interface allows applications to find out the types of formats that the capture pin can connect with, and it allows the application to set (or hard wire) the exact format that it wants the capture pin to connect with—as long as it supports that format.

Once the application determines that this is a capture pin, it tries to find out what type of format it currently supports. Typically, the current format information (image format, compression type, and image size) is set by the user and is stored in the system registry. The IAMStreamConfig interface provides the *GetFormat()* and *SetFormat()* functions in order to allow the application to manipulate the capture format of the filter.

25.5.4 The IAMBufferNegotiation Interface—Optional

To allow applications more control over the buffer allocation used for the pin, the capture pin can expose this interface. The IAMBufferNegotiation interface provides the function *SuggestAllocatorProperties(),* which an application can call with its own allocator properties (number of buffers, size of buffers, alignment, and so forth); the capture pin uses this allocator when it allocates the shared buffer with the downstream filter. An application can also get the current allocator properties by calling the *GetAllocatorProperties()* function.

```
HRESULT CCaptureStream::SuggestAllocatorProperties(const ALLOCATOR_PROPERTIES *pprop)
{
    // to make sure we're not in the middle of connecting
    CAutoLock lock(m_pFilter->pStateLock());
    CAutoLock l(&m_cSharedState);

    m_propSuggested = *pprop;
    return NOERROR;
}


HRESULT CCaptureStream::GetAllocatorProperties(ALLOCATOR_PROPERTIES *pprop)
{
    // to make sure we're not in the middle of connecting
    CAutoLock lock(m_pFilter->pStateLock());
    CAutoLock l(&m_cSharedState);

     *pprop = m_propActual;
    return NOERROR;
}
```

25.5.5 The IAMVideoCompression Interface

Applications use this interface to extract information about the filter and to set a variety of parameters. The filter overrides the *IAMVideoCompression::GetInfo()* function to provide information about itself such as its name, version, key frame rate, quality, and so forth. The IAMVideoCompression interface also provides functions to set some of these parameters, such as *put_KeyFrameRate(), put_WindowSize(), put_Quality(), put_PFramesPerKeyFrame(),* and so forth. Notice that we've in-lined these functions in the declaration of the CCaptureStream class above.

```
HRESULT CCaptureStream::GetInfo(LPWSTR pszVersion, int *pcbVersion, LPWSTR pszDescription, int
*pcbDescription, long *pDefaultKeyFrameRate, long *pDefaultPFramesPerKey, double
*pDefaultQuality, long *pCapabilities)
{
    if (pCapabilities)
        *pCapabilities = 0;
    if (pDefaultKeyFrameRate)
        *pDefaultKeyFrameRate = 0;
    if (pDefaultPFramesPerKey)
        *pDefaultPFramesPerKey = 0;
    if (pDefaultQuality)
        *pDefaultQuality = 0;

    // we can give them a driver name and version
    if (pszVersion && pcbVersion)
         lstrcpynW(pszVersion, m_szVersion, *pcbVersion / 2);
    if (pszDescription && pcbDescription)
         lstrcpynW(pszDescription, m_szName, *pcbDescription / 2);
```

```
    // return the number of bytes this unicode string is, incl. NULL char
    if (pcbVersion)
        *pcbVersion = lstrlenW(m_szVersion) * 2 + 2;// remember, this is UNICODE
    if (pcbDescription)
        *pcbDescription = lstrlenW(m_szName) * 2 + 2;

    return NOERROR;
}
```

25.5.6 Reporting Dropped Frames Using IAMDroppedFrames

The capture pin can also expose the IAMDroppedFrames interface in order to inform the application of the number of dropped frames. The capture filter implements the *GetNumDropped()* function, which reports the number of frames dropped for one reason or another. It also implements the function *GetNumNotDropped(),* which indicates how many frames were delivered successfully.

```
HRESULT CCaptureStream::GetNumDropped(long *plDropped)
{
    if (plDropped == NULL)
       return E_POINTER;

    *plDropped = (long)m_uiFramesSkipped;
    return NOERROR;
}


HRESULT CCaptureStream::GetNumNotDropped(long *plNotDropped)
{
    if (plNotDropped == NULL)
       return E_POINTER;

    *plNotDropped = (long)m_uiFramesDelivered;
    return NOERROR;
}
```

The capture filter also reports the average size per frame by responding to the *GetAverageFrameSize()* function as shown below.

```
HRESULT CCaptureStream::GetAverageFrameSize(long *plAverageSize)
{
    if (plAverageSize == NULL)
       return E_POINTER;

    *plAverageSize = m_uiFramesDelivered ?
    (long)(m_llTotalFrameSize / m_uiFramesDelivered) : 0;

    return NOERROR;
}
```

25.6 Writing a Video Capture Application

Typically, capture applications need to have more control over the settings and characteristics of the captured image. The capture application must figure out which capture device it should use and then load the appropriate capture filter, select the appropriate capture format, and render the output pins (capture, preview, or overlay) based on the user's selection.

A capture application performs the following steps to configure the capture filter:

- It enumerates all the installed capture filters and loads the appropriate one based on the user's selection or some other criteria.
- It retrieves pointers to the interfaces it wants to use, such as IAMDroppedFrames, IAMVideoCompression, IAMStreamConfig, IAMVFWCaptureDialog, and so forth.
- It starts a preview session—if supported by the filter and enabled by the user.
- It starts a capture session when the user requests a capture. Here the application sets the output filename, the frame rate, and the image format.
- An application can retrieve the number of dropped frames, receive events from the capture filter, or display the VFW dialog boxes.

25.6.1 Enumerating Installed Capture Filters

To start off, let's see how we can figure out which devices are installed in the system and decide which one to load. To do that, we must first create an instance of the ICaptureGraphBuilder interface.

```
CoCreateInstance(
    (REFCLSID)CLSID_CaptureGraphBuilder,
    NULL,
    CLSCTX_INPROC,
    (REFIID)IID_ICaptureGraphBuilder,
    (void **)&gcap.pBuilder);        // the interface is returned here
```

DirectShow lists capture filters in the system registry under the *CLSID_VideoInputDeviceCategory* key. In order to figure out which capture adapter to use, we enumerate all the capture devices in the system and then load the one that we're interested in. This process is called binding to the object.

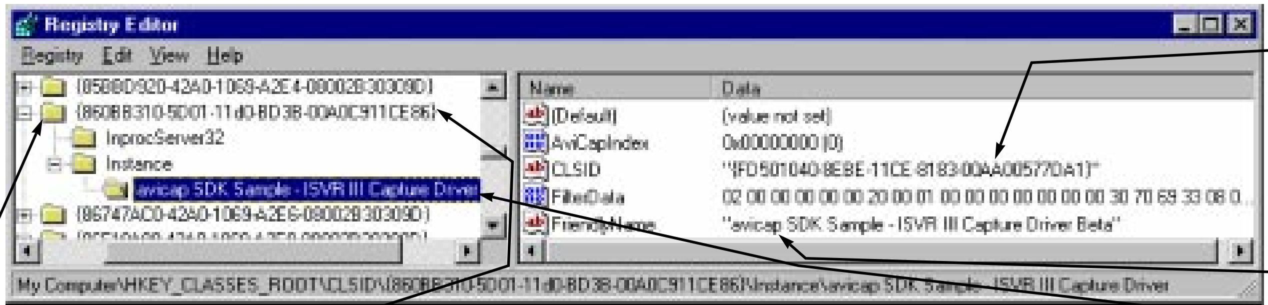


Figure 25-6 DirectShow registry setting for the ISVRIII capture filter.

This is the value of *CLSID_VideoInputDeviceCategory*. It points to the list of video capture devices in the system.

To enumerate the installed capture devices in the system, we must create a device enumerator. To do that, we call the *CoCreateInstance()* function with the IID_ICreateDevEnum as a parameter. We then call the *ICreateDevEnum::CreateClassEnumerator()* function with the *CLSID_VideoInputDeviceCategory* class ID in order to create a video capture device enumerator. This is returned in the pEm variable. The video device category is shown in the Registry Editor snapshot below.

```
ICreateDevEnum *pCreateDevEnum;
CoCreateInstance(CLSID_SystemDeviceEnum, NULL, CLSCTX_INPROC_SERVER,
        IID_ICreateDevEnum, (void**)&pCreateDevEnum);

IEnumMoniker *pEm;
pCreateDevEnum->CreateClassEnumerator(CLSID_VideoInputDeviceCategory, &pEm, 0);
 pCreateDevEnum->Release();

VARIANT var;
ULONG cFetched;
 IMoniker *pM;
 IPropertyBag *pBag;

pEm->Reset();
gcap.pVCap = NULL;


while(pEm->Next(1, &pM, &cFetched) == S_OK) {
   // this is the one we want.  Get its name, and instantiate it.
   if ((int)uIndex == gcap.iVideoDevice) {
       gcap.achFriendlyName[0] = 0;
      pM->BindToStorage(0, 0, IID_IPropertyBag, (void **)&pBag);

       var.vt = VT_BSTR;
      pBag->Read(L"FriendlyName", &var, NULL);
    WideCharToMultiByte(CP_ACP, 0, var.bstrVal, -1, gcap.achFriendlyName, 80, NULL, NULL);
       SysFreeString(var.bstrVal);
       pBag->Release();

      pM->BindToObject(0, 0, IID_IBaseFilter, (void**)&gcap.pVCap);
    pM->Release();
       break;
   }
   pM->Release();
   uIndex++;
}
 pEm->Release();
```

Now that we have access to the video capture device enumerator, we call its *Reset()* function to start from the top of the list and then call the *Next()* function repeatedly until we find the capture device that we're interested in. The *Next()* function returns a moniker object that we can use to create a property bag that allows us to retrieve information about the device. In this case, the property bag is pointing to the device key in the system registry as shown in the snapshot. We can use the property bag to read the "Friendly-Name" of the device and figure out if this is the device we're looking for. Once we find the capture device that we want to load, we can call the *Bind-ToStorage()* function to get a pointer to an IBaseFilter interface. The function creates the filter and returns a pointer to it in the last parameter.

Now that we have a pointer to the filter, we can retrieve all the interfaces that we need to access. To do that, we use the *ICaptureGraphBuilder::FindInterface()* and specify the `PIN_CATEGORY_CAPTURE` in addition to the GUID of the interface that we want to retrieve. In our case, we look for the IAM-DroppedFrames, IAMVideoCompression, IAMStreamConfig, and IAMVF-WCaptureDialogs interfaces. We'll use these interfaces later in the sample.

```
    // we use this interface to get the number of captured and dropped
frames
    gcap.pBuilder->FindInterface(&PIN_CATEGORY_CAPTURE, gcap.pVCap,
        IID_IAMDroppedFrames, (void **)&gcap.pDF);

    // we use this interface to get the name of the driver
    gcap.pBuilder->FindInterface(&PIN_CATEGORY_CAPTURE, gcap.pVCap,
        IID_IAMVideoCompression, (void **)&gcap.pVC);

    gcap.pBuilder->FindInterface(&PIN_CATEGORY_CAPTURE, gcap.pVCap,
        IID_IAMStreamConfig, (void **)&gcap.pVSC);

    // we use this interface to bring up the 3 dialogs
    // NOTE:  Only the VFW capture filter supports this.  This app only
brings
    // up dialogs for legacy VFW capture drivers, since only those have
dialogs
    hr = gcap.pBuilder->FindInterface(&PIN_CATEGORY_CAPTURE, gcap.pVCap,
        IID_IAMVFWCaptureDialogs, (void **)&gcap.pDlg);
```

Next, we ask the capture filter for the current setting of the capture driver and set our window size accordingly.

```
  AM_MEDIA_TYPE *pmt;
  gcap.pVSC->GetFormat(&pmt);        // current capture format
  // resize our window to the new capture size
  ResizeWindow(HEADER(pmt->pbFormat)->biWidth,
     HEADER(pmt->pbFormat)->biHeight);
  DeleteMediaType(pmt);
```

25.6.2 Starting a Preview Session

At this stage, we've only figured out which capture filter to load and only created the capture filter object. Notice that the filter itself has not been loaded into a filter graph and that none of the filter pins have been rendered. To start a preview session on the preview pin, we must first create a filter graph and load the capture filter into it. We call the *QzCreateFilterObject()* function to create a filter graph and then inform the capture graph builder that we want to use this filter graph to load the capture filter. We then call the *IGraphBuilder::AddFilter()* to add the capture filter into the newly created filter graph.

```
QzCreateFilterObject(CLSID_FilterGraph, NULL, CLSCTX_INPROC,
    IID_IGraphBuilder, (LPVOID *)&gcap.pFg);
gcap.pBuilder->SetFiltergraph(gcap.pFg);
gcap.pFg->AddFilter(gcap.pVCap, NULL);
```

To render the preview pin on the capture filter, we call the *ICaptureGraphBuilder::RenderStream()* function with the `PIN_CATEGORY_PREVIEW` as a parameter. Notice that we didn't use the *RenderFile()* function to render the filter graph since we don't have a physical file to render. The *RenderStream()* function is the equivalent of the *RenderFile()* function; it figures out which filters it needs to load in order to display the captured video in a window.

```
gcap.pBuilder->RenderStream(&PIN_CATEGORY_PREVIEW, NULL,
     gcap.pVCap, NULL, NULL);
```

Now the filter graph is ready to run. To run the capture filter, we retrieve an IMediaControl interface object and call its *Run()* function. The capture filter, in turn, starts delivering digitized images from the capture adapter and delivers them to the downstream filters—probably a color converter and a video renderer.

```
// run the graph
  IMediaControl *pMC = NULL;
gcap.pFg->QueryInterface(IID_IMediaControl, (void **)&pMC);
pMC->Run();
pMC->Release();
```

25.6.3 Starting a Capture Session

Notice that we're assuming that the preview pin is already connected—as we've just discussed. We are starting just after the capture filter was created and the window size was set to match the current filter format.

Building a capture filter graph is similar to building the preview filter graph except that the capture filter graph must specify an output file to store the capture data into. In order to capture a video clip into a file, we need to get the filename of the file where the stream will be saved. In our sample the filename has been already stored in the gcap.szCaptureFile variable.

For a smoother capture, it's typically better to pre-allocate the space for the file on the hard drive before starting the capture. Allocating new sectors on the hard drive during a capture session consumes a lot of time and causes frames to drop. With pre-allocation, you save this step and write directly to the empty space in the file. To pre-allocate a file, we call the *ICaptureGraph-Builder::AllocCapFile()* function, which accepts the filename and the size of the file to allocate—in bytes.

```
WCHAR wach[_MAX_PATH];

// if the file does not already exist, preallocate it..
if (OpenFile(gcap.szCaptureFile, &os, OF_EXIST) == HFILE_ERROR) {
    MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED, gcap.szCaptureFile, -1, wach, _MAX_PATH);
    gcap.pBuilder->AllocCapFile(wach, (DWORDLONG)gcap.wCapFileSize * 1024L * 1024L);
}
```

So far, we've only allocated the file on the hard drive, but we have not built the capture filter graph. To do that, we call the *ICaptureFilterBuilder::Set-OutputFileName()* function, which loads two filters: an *AVI mixer* and a *file writer* filter. The AVI mixer mixes audio and video streams into the AVI file format; this is the only supported format as of the beta 4 release of the DirectShow SDK. The file writer takes the AVI data and writes it out to the output file. The *SetOutputFileName()* function returns a pointer to the AVI mixer, gcap.pRender, and the file writer, gcap.pSink (of type IFilterSinkFil-ter). You can use the `pRender` filter as the renderer parameter when you call the *ICaptureFilterGraph::RenderStream()* function. You can use the pSink pointer to change the name of the output file by calling the *IFilterSinkFil-ter::SetFileName()* function.

```
  GUID guid = MEDIASUBTYPE_Avi;
  gcap.pBuilder->SetOutputFileName(&guid, wach, &gcap.pRender,
&gcap.pSink);
```

Notice that the newly loaded filters, AVI mixer and file writer, were loaded into the filter graph, but they have not been connected. Notice also that the capture filter is not even loaded into this specific filter graph. To add the capture filter, we retrieve a pointer to the current filter graph, where the AVI mixer and file writer filters were loaded, and then call the *AddFilter()* function to add the capture filter. Still nothing is connected.

```
  gcap.pBuilder->GetFiltergraph(&gcap.pFg);
  gcap.pFg->AddFilter(gcap.pVCap, NULL);
```

Finally we call the *RenderStream()* function to render the capture output pin of the capture filter. Notice that we loaded the AVI mixer and file writer filters before we rendered the capture pin on the capture filter. If we didn't do that, the *RenderStream()* function would have created a filter graph that uses the fact that the capture pin can produce video data and then adds a video renderer to display the images to the screen—rather than saving it to a file. For example, if the capture pin is producing YUV9 data, the *RenderStream()* would have loaded a YUV9 color converter filter and a video renderer and displayed the image on the screen—just like it did with the preview pin.

```
  gcap.pBuilder->RenderStream(&PIN_CATEGORY_CAPTURE, NULL,
     gcap.pVCap, NULL, gcap.pRender);
```

Now that the filter graph is connected, we can start the capture session by running the filter graph.

```
  // run the graph
  IMediaControl *pMC = NULL;
  gcap.pFg->QueryInterface(IID_IMediaControl, (void **)&pMC);
  pMC->Run();
  pMC->Release();
```

The capture filter graph now receives digitized images from the capture adapter and stores them in the specified file.

25.6.4 Setting the Frame Rate

To set the capture frame rate, we first get the current format of the filter, and we change only the frame rate element of the `VIDEOINFOHEADER` structure. Notice that the frame rate is specified by the average time per frame (in milliseconds).

```
gcap.pVSC->GetFormat(&pmt);
VIDEOINFOHEADER *pvi = (VIDEOINFOHEADER *)pmt->pbFormat;
pvi->AvgTimePerFrame = (LONGLONG)(10000000 / gcap.FrameRate);
gcap.pVSC->SetFormat(pmt);
DeleteMediaType(pmt);
```

25.6.5 How Well Are We Capturing?

You can use the IAMDroppedFrames interface to figure out how well you're capturing. For example, you can call the *GetNumDropped()* function to figure out how many frames you've dropped; you can call the *GetNumNotDropped()* function to figure out how many frames were delivered; and you can call the *GetAverageFrameSize()* function to figure out the average size of the frames. You typically use this information to, perhaps, change the quality of the captured frames, or to change the frame rate, so you can drop less frames.

```
gcap.pDF->GetNumDropped(&lDropped);
gcap.pDF->GetNumNotDropped(&lNot);
gcap.pDF->GetAverageFrameSize(&lAvgFrameSize);
```

25.6.6 Handling Events of the Filter Graph

To accept notification messages from the filter graph, you need to register a notification message with the event manager. To do that, you must first get an IMediaEventEx object and call its *SetNotifyWindow()* function to register a notification message.

```
gcap.pFg->QueryInterface(IID_IMediaEventEx, (void**)&gcap.pME);
gcap.pME->SetNotifyWindow((LONG)ghWndApp, WM_FGNOTIFY, 0);
```

When the filter graph sends a notification message, it calls the Win32 function *SendMessage()* for the specified window and the `WM_FGNOTIFY` message. The window handler responds to the message and handles the event accordingly. The application calls *IMediaEventEx::GetEvent()* to retrieve the event and handles it accordingly.

```
case WM_FGNOTIFY:
    // uh-oh, something went wrong while capturing - the filtergraph
    // will send us events like EC_COMPLETE, EC_USERABORT and the one
    // we care about, EC_ERRORABORT.
    if (gcap.pME && gcap.fCapturing) {
        LONG event, l1, l2;
        while (gcap.pME->GetEvent(&event, &l1, &l2, 0) == S_OK) {
    if (event == EC_ERRORABORT) {
        // pME will go away in BuildPreviewGraph
        gcap.pME->FreeEventParams(event, l1, l2);
        StopCapture();
        if (gcap.fWantPreview) {
    BuildPreviewGraph();
    StartPreview();
        }
        ErrMsg("ERROR during capture - disk possibly full");
        break;
    }
    gcap.pME->FreeEventParams(event, l1, l2);
        }
    }
    break;
```

25.6.7 Showing VFW Format and Source Dialogs

As we've mentioned earlier, an application can ask the default DirectShow capture filter (the VFW wrapper) to display the configuration dialog boxes for the VFW driver. To display the Format dialog box, the application first calls the *IAMVFWCaptureDialog::HasDialog()* function with `VFWCAPTUREDIALOG_FORMAT` as a parameter. If the dialog box is supported, you can call the *IAMVFWCaptureDialog::ShowDialog()* function with `VFWCAPTUREDIALOG_FORMAT` and the application window handle. Now, since the user could have changed the format, you need to reset the size of your window accordingly.

```
if(!gcap.pDlg->HasDialog(VFWCaptureDialog_Format)) {
    if (gcap.fPreviewing)
        StopPreview();
    gcap.pDlg->ShowDialog(VFWCaptureDialog_Format, ghwndApp);

    // current capture format
    AM_MEDIA_TYPE *pmt;
    gcap.pVSC->GetFormat(&pmt);
    // resize our window to the new capture size
    ResizeWindow(HEADER(pmt->pbFormat)->biWidth, HEADER(pmt->pbFormat)->biHeight);
    DeleteMediaType(pmt);

    if (gcap.fWantPreview)
        StartPreview();
}
```

Finally, you can do the same thing for displaying the Source Input dialog box as follows.

```
if(!gcap.pDlg->HasDialog(VFWCaptureDialog_Source)) {
    if (gcap.fPreviewing)
    StopPreview();
    gcap.pDlg->ShowDialog(VFWCaptureDialog_Source, ghwndApp);
    if (gcap.fWantPreview)
    StartPreview();
}
```

What Have You Learned?

This late-breaking chapter introduced you to the latest features of DirectShow, namely capture and compression. You were first introduced to the concept of video capture using the Video for Windows (VFW) architecture. You then learned about the Windows Device Model driver model and how it relates to DirectShow. You then stepped through the DirectShow sample video capture filter where you learned about the capture and preview pins. Finally, you learned how to access a capture filter from your application.

Since we wrote this chapter while DirectShow was still in beta 4, we highly recommend that you refer to the documentation of the final release of DirectShow, which should be available by the time you're reading this chapter.