HRESULT get_FramesDrawn(
int *pcFramesDrawn*
);

Parameters

pcFramesDrawn
Number of frames drawn since streaming started.

Return Values

Returns an HRESULT value.

Previous Home Topic Contents Index Next

IQualProp::get_FramesDroppedInRenderer

IQualProp Interface

Retrieves the number of frames dropped by the renderer.

HRESULT get_FramesDroppedInRenderer(
int *pcFrames*
);

Parameters

pcFrames
Number of frames dropped by the renderer.

Return Values

Returns an HRESULT value.

Remarks

The property page uses this method to retrieve data from the renderer.

Previous Home Topic Contents Index Next

IQualProp::get_Jitter

IQualProp Interface

Expresses the average time between successive frames delivered to the video renderer.

HRESULT get_Jitter(
int **piJitter*
);

Parameters

piJitter
Standard deviation, in milliseconds, of the interframe time.

Return Values

Returns an HRESULT value.

Previous Home Topic Contents Index Next

IQueueCommand Interface

The **IQueueCommand** interface provides a way to defer commands and property changes. The deferred command mechanism allows filters themselves to handle deferred commands. When they do not, the filter graph manager queues the command until the requested time and then calls the method on the filter (this would result in coarse rather than accurate synchronization). Note that a filter that does handle deferred commands must make them apply to data appearing at that time. Thus, a contrast filter asked to change the contrast at time x must ensure that it applies the change when processing data time-stamped to be rendered at time x; these samples will be processed by the filter somewhat before time x.

The **IQueueCommand** interface provides two methods, InvokeAtStreamTime, which queues commands at stream time, and InvokeAtPresentationTime, which queues commands at presentation time. Both return an IDeferredCommand interface to the queued command, by which the application can cancel the command, set a new presentation time for it, or get back an estimate of the likelihood of the filter graph manager being able to run the command on time.

Both presentation time and stream time commands will run once, and then be removed from the queue. Both the queue and the application will hold a reference count on the object (represented to the application by the IDeferredCommand interface), and the object will not be destroyed until both are released. Similarly, calling IUnknown::Release on the **IDeferredCommand** interface is not sufficient to cancel the command, because the queue also holds a reference count.

Rather than add optional stream time and presentation time constraints to every method and property on every control interface, the application uses IDispatch to provide a single interface where these time parameters can be specified. **IQueueCommand** provides InvokeAtStreamTime and InvokeAtPresentationTime methods that are similar in style to the IDispatch::Invoke method.

Filters can implement **IQueueCommand** themselves. In this case, they parse the command and queue it for action when the relevant samples arrive or when the reference clock reaches the correct point, as appropriate. For filters that do not support this, the filter graph manager will run the command when the reference clock reaches the specified time, regardless of the samples being processed at the filter.

When to Implement

This method is implemented by the filter graph manager to allow queuing of deferred commands.

When to Use

Applications can use this interface, along with the IDeferredCommand interface, to queue commands for deferred processing.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IQueueCommand methods	Description
InvokeAtStreamTime	Queues a method or property change for execution at a specified stream time (that is, presentation time relative to the current stream time offset).
InvokeAtPresentationTime	Queues a method or property change for execution at a specified presentation time.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IQueueCommand::InvokeAtPresentationTime

IQueueCommand Interface

Queues a method or property change for execution at a specified presentation time.

HRESULT InvokeAtPresentationTime(
IDeferredCommand * *pCmd*,
REFTIME *time*,
GUID* *iid*,
long *dispidMember*,
short *wFlags*,
long *cArgs*,
VARIANT **pDispParams*,
VARIANT **pvarResult*,
short **puArgErr*
);

Parameters

pCmd
[out] Pointer to the place to return an interface on the deferred command if it is successfully created.
time
[in] Time at which to invoke the command.
iid
[in] Interface to be called.
dispidMember
[in] Method or property to call on the interface.
wFlags
[in] Method or property flag.
cArgs
[in] Number of arguments on *pDispParams*.
pDispParams
[in] Parameters to this method.
pvarResult
[in,out] Return value.
puArgErr
[out] Index to the arguments in error.

Return Values

Returns an HRESULT value.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IQueueCommand::InvokeAtStreamTime

IQueueCommand Interface

Queues a method or property change for execution at a specified stream time (that is, presentation time relative to the current stream time offset).

HRESULT InvokeAtStreamTime(
IDeferredCommand ** *pCmd*,
REFTIME *time*,
GUID *iid*,
long *dispidMember*,
short *wFlags*,
long *cArgs*,
VARIANT *pDispParams*,
VARIANT *pvarResult*,
short *puArgErr*
);

Parameters

pCmd
[out] Pointer to the place to return an interface on the deferred command if it is successfully created.
time
[in] Time at which to invoke the command.
iid
[in] Interface to be called.
dispidMember
[in] Method or property to call on the interface.
wFlags
[in] Method or property flag.
cArgs
[in] Number of arguments in *pDispParams*.
pDispParams
[in] Parameters to this method.
pvarResult
[in, out] Return value of the called method.
puArgErr
[out] Index to the arguments in error.

Return Values

Returns an HRESULT value.

Remarks

Run this command to affect the presentation that occurs after the specified stream time. The interface IID is an interface that can be obtained by calling IUnknown::QueryInterface on this same IQueueCommand interface.

Previous Home Topic Contents Index Next

IReferenceClock Interface

The **IReferenceClock** interface represents a system reference clock to be implemented by a filter in the filter graph and used by other filters.

When to Implement

Implement this interface if you are writing a filter that generates a system reference clock. Typically, this applies to audio renderer filters because audio sound boards usually contain a reference clock. Use the CBaseReferenceClock class to implement this interface.

When to Use

Use this interface on any filter to obtain reference clock notifications for a duration of elapsed time (both singular and repetitive), or to retrieve the current time.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IReferenceClock methods	Description
GetTime	Gets the current time.
AdviseTime	Requests an asynchronous notification that a duration has elapsed.
AdvisePeriodic	Requests an asynchronous, periodic notification that a duration has elapsed.
Unadvise	Cancels a request for notification.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IReferenceClock::AdvisePeriodic

IReferenceClock Interface

Requests an asynchronous, periodic notification that a duration has elapsed.

HRESULT AdvisePeriodic(
REFERENCE_TIME *rtStartTime*,
REFERENCE_TIME *rtPeriodTime*,
HSEMAPHORE *hSemaphore*,
DWORD * *pdwAdviseCookie*
);

Parameters

rtStartTime
[in] Time the notification should begin.
rtPeriodTime
[in] Duration between notifications.
hSemaphore
[in] Handle of a semaphore through which to advise.
pdwAdviseCookie
[out] Used to identify this call to **AdvisePeriodic** in the future; for example, to cancel it.

Return Values

Returns an HRESULT value that depends on the implementation. **HRESULT** can be one of the following standard constants, or other values not listed:

Value	Meaning
E_FAIL	Failure.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method isn't supported.
S_OK or NOERROR	Success.

Remarks

When the time indicated by *rtStartTime* is reached, the semaphore whose handle is set as *hSemaphore* will be released. Thereafter, the semaphore will be released repetitively with a period of *rtPeriodTime*.

See Also

IReferenceClock::Unadvise, CBaseReferenceClock::AdvisePeriodic

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IReferenceClock::AdviseTime

IReferenceClock Interface

Requests an asynchronous notification that a duration has elapsed.

```
HRESULT AdviseTime(
  REFERENCE_TIME rtBaseTime,
  REFERENCE_TIME rtStreamTime,
  HEVENT hEvent,
  DWORD * pdwAdviseCookie
  );
```

Parameters

rtBaseTime
[in] Base reference time.
rtStreamTime
[in] Stream offset time.
hEvent
[in] Handle of an event through which to advise.
pdwAdviseCookie
[out] Destination of the token.

Return Values

Returns an HRESULT value that depends on the implementation. **HRESULT** can be one of the following standard constants, or other values not listed:

Value	Meaning
E_FAIL	Failure.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method isn't supported.
S_OK or NOERROR	Success.

Remarks

When the time *rtBaseTime*+*rtStreamTime* is reached, the event whose handle is *hEvent* will be set. If the time has already passed, the event will be set immediately.

See Also

IReferenceClock::Unadvise

Previous Home Topic Contents Index Next

IReferenceClock::GetTime

IReferenceClock Interface

Retrieves the current time. **REFERENCE_TIME** is a LONGLONG type and loosely represents the number of 100-nanosecond units that have elapsed since some fixed start time. See Characteristics of a Reference Clock for other requirements on the reference clock.

HRESULT GetTime(
REFERENCE_TIME * *pTime*
);

Parameters

pTime
[out] Current time.

Return Values

Returns an HRESULT value that depends on the implementation. **HRESULT** can be one of the following standard constants, or other values not listed:

Value	Meaning
E_FAIL	Failure.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method isn't supported.
S_OK or NOERROR	Success.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IReferenceClock::Unadvise

IReferenceClock Interface

Cancels a request for notification.

HRESULT Unadvise(

DWORD *dwAdviseCookie*
);

Parameters

dwAdviseCookie
[in] Request to cancel.

Return Values

Returns an HRESULT value that depends on the implementation. **HRESULT** can be one of the following standard constants, or other values not listed:

Value	Meaning
E_FAIL	Failure.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method isn't supported.
S_OK or NOERROR	Success.

See Also

IReferenceClock::AdviseTime, IReferenceClock::AdvisePeriodic

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IRegFilterInfo Interface

The **IRegFilterInfo** interface provides access to filters in the registry and allows a registered filter to be added to the filter graph.

When to Implement

IRegFilterInfo is implemented by the filter graph manager for use by Automation client applications, such as Microsoft® Visual Basic®.

When to Use

Use this interface when it is exposed by an Automation client to query the names of filters in a collection of registry filters, and to add specific filters to the filter graph. A collection of IRegFilterInfo interfaces is returned by the IMediaControl::get_RegFilterCollection method.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IDispatch methods	Description
GetTypeInfoCount	Determines whether there is type information available for this dispinterface.
GetTypeInfo	Retrieves the type information for this dispinterface if GetTypeInfoCount returned successfully.
GetIDsOfNames	Converts text names of properties and methods (including arguments) to their corresponding DISPIDs.
Invoke	Calls a method or accesses a property in this dispinterface if given a DISPID and any other necessary parameters.

IRegFilterInfo methods	Description
get_Name	Retrieves the name of the filter.
Filter	Creates an instance of this filter and adds it to the filter graph.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IRegFilterInfo::Filter

IRegFilterInfo Interface

Creates an instance of this filter and adds it to the filter graph.

```
HRESULT Filter(
  IDispatch **ppUnk
  );
```

Parameters

ppUnk
[out] IFilterInfo interface for the added filter.

Return Values

Returns an HRESULT value.

Remarks

Use the IRegFilterInfo::get_Name method (Name property in Visual Basic) to find the filter by comparing names in a collection of IRegFilterInfo interfaces.

Previous Home Topic Contents Index Next

IRegFilterInfo::get_Name

IRegFilterInfo Interface

Retrieves the name of the filter.

HRESULT get_Name(
BSTR * *strName*
);

Parameters

strName
[out, retval] Name of the filter.

Return Values

Returns an HRESULT value.

Remarks

Typically, a Visual Basic application will use the `For Each...Next` syntax on a collection of IRegFilterInfo interfaces and check the name of each filter in the registry until it finds the one it wants to add. It can then add the filter to the filter graph by using the IRegFilterInfo::Filter method.

Previous Home Topic Contents Index Next

IResourceConsumer Interface

The **IResourceConsumer** interface implements a resource consumer that requests resources from a resource manager that supports the IResourceManager interface.

When to Implement

Implement this interface on any object that requests resources from a resource manager. (The filter graph manager acts as a resource manager for Microsoft® DirectShow™.) After implementing this interface, the object can register resources that it wants to use. It passes a pointer to this interface when it does this so that the resource manager can use methods on this interface to inform the object that a resource is available, or to release a resource that it is using.

When to Use

A resource manager that implements the IResourceManager interface calls methods on this interface.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IResourceManager methods	Description
AcquireResource	Notifies the resource consumer that a resource might be acquired.
ReleaseResource	Requests the resource consumer to release the specified resource.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IResourceConsumer::AcquireResource

IResourceConsumer Interface

Notifies the resource consumer that a resource might be acquired.

```
HRESULT AcquireResource(
  LONG idResource
  );
```

Parameters

idResource
[in] Resource identifier of the resource to be acquired.

Return Values

Returns one of the following values.

Value	Meaning
S_OK	Consumer has successfully acquired the resource.
S_FALSE	Consumer has not acquired the resource but will use IResourceManager::NotifyAcquire when it does.
VFW_S_RESOURCE_NOT_NEEDED	Consumer no longer needs the resource.
Error Value	Consumer tried to acquire the resource but failed.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IResourceConsumer::ReleaseResource

IResourceConsumer Interface

Requests the resource consumer to release the specified resource.

HRESULT ReleaseResource(
LONG *idResource*
);

Parameters

idResource
[in] Resource identifier to be released.

Return Values

Returns S_OK if the consumer has released it and requires it again when it becomes available, or S_FALSE if the consumer has not released it but will use IResourceManager::NotifyRelease when it does.

Previous Home Topic Contents Index Next

IResourceManager Interface

The **IResourceManager** interface implements a resource manager to resolve contentions for named resources.

When to Implement

Implement this interface on any object that performs the services of a resource manager. The filter graph manager acts as a resource manager for Microsoft® DirectShow™ and delegates to any existing system-wide resource manager. The filter graph manager implements the methods on this interface.

When to Use

Use this interface if your object requires resources that other objects are likely to use. The wave renderer uses this interface to resolve contentions for the wave-output device to enable sound to follow focus.

An object can use the resource manager supporting this interface to resolve possible contention between existing resources. This is carried out by registering the resource with the interface and then requesting it from this interface whenever needed.

Use this interface if your object detects user focus changes that might affect resource usage. Notifying the resource manager of a change of focus will cause the resource manager to switch contended resources to the objects that have the focus of the user.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IResourceManager methods	Description
Register	Registers a single named resource with the resource manager.
RegisterGroup	Registers a named resource group with the resource manager.
RequestResource	Requests the use of a given registered resource.
NotifyAcquire	Notifies the resource manager that an attempt to acquire a resource has completed.
NotifyRelease	Notifies the resource manager that a resource consumer has released a resource.
CancelRequest	Cancels the request for a resource.
SetFocus	Notifies the resource manager that a specified object has been given the focus of the user.
ReleaseFocus	Sets the focus object to NULL in the resource manager if the object of the current focus object is the one specified in this method.

Previous Home Topic Contents Index Next

IResourceManager::CancelRequest

IResourceManager Interface

Cancels the request for a resource.

HRESULT CancelRequest(
LONG *idResource*,
IResourceConsumer* *pConsumer*
);

Parameters

idResource
[in] Resource identifier of a pending request.
pConsumer
[in] IResourceConsumer interface that made the request.

Return Values

Returns an HRESULT value that depends on the implementation. **HRESULT** can be one of the following standard constants, or other values not listed:

Value	Meaning
E_FAIL	Failure.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method isn't supported.
S_OK or NOERROR	Success.

Remarks

This method should be called when the IResourceConsumer object that requested the resource has not received it and no longer requires it. If it has already received the resource, it should use the IResourceManager::NotifyRelease method.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IResourceManager::NotifyAcquire

IResourceManager Interface

Notifies the resource manager that an attempt to acquire a resource has completed.

HRESULT NotifyAcquire(
LONG *idResource*,
IResourceConsumer* *pConsumer*,
HRESULT *hr*
);

Parameters

idResource
[in] Token for the registered resource.
pConsumer
[in] IResourceConsumer interface of the object requesting the resource.
hr
[in] Success of the acquisition; S_OK if the resource was acquired, or an error value if not.

Return Values

Returns an HRESULT value that depends on the implementation. **HRESULT** can be one of the following standard constants, or other values not listed:

Value	Meaning
E_FAIL	Failure.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method isn't supported.
S_OK or NOERROR	Success.

Remarks

Use this method after an IResourceConsumer::AcquireResource method returns an S_FALSE value, indicating that the acquisition will be asynchronous (that is, handled by a callback mechanism). If the *hr* parameter is S_OK, the resource manager will assume that the resource is now held by the caller. If the *hr* parameter is anything other than S_OK, the resource manager will assume that the attempt to acquire the resource failed and will reassign the resource elsewhere.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IResourceManager::NotifyRelease

IResourceManager Interface

Notifies the resource manager that IResourceConsumer has released a resource.

HRESULT NotifyRelease LONG *idResource*,

IResourceConsumer* *pConsumer*,
BOOL *bStillWant*
);

Parameters

idResource
[in] Resource token.
pConsumer
[in] Object releasing the resource.
bStillWant
[in] Flag specifying whether the resource is still required or not.

Return Values

Returns an HRESULT value that depends on the implementation. **HRESULT** can be one of the following standard constants, or other values not listed:

Value	Meaning
E_FAIL	Failure.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method isn't supported.
S_OK or NOERROR	Success.

Remarks

Use this method in response to an IResourceConsumer::ReleaseResource method, or when you have finished using the resource. The *bStillWant* parameter should be set TRUE if you still want the resource when it is next available, or FALSE if you no longer want the resource.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IResourceManager::Register

IResourceManager Interface

Registers a single named resource with the resource manager.

HRESULT Register(
LPCWSTR *pName*,
LONG *cResource*,
LONG* *plToken*
);

Parameters

pName
[in] Named resource.
cResource
[in] Number of resources.
plToken
[out] Returned token identifying the resource to be used in additional calls.

Return Values

Returns an HRESULT value that depends on the implementation. **HRESULT** can be one of the following standard constants, or other values not listed:

Value	Meaning
E_FAIL	Failure.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method isn't supported.
S_OK or NOERROR	Success.

Remarks

This method registers a named resource, which can contain a number of resources, and returns a token to be used when requesting this resource. It is not an error if the resource is already registered; if the number in the *cResource* parameter is less than what is already registered, resources will be deallocated to the new count. To unregister the resource, pass a count of zero in *cResource*.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IResourceManager::RegisterGroup

IResourceManager Interface

Registers a named resource group with the resource manager.

```
HRESULT RegisterGroup(
  LPCWSTR pName,
  LONG cResource,
  LONG* palTokens,
  LONG* plToken
  );
```

Parameters

pName
[in] Named resource group.
cResource
[in] Number of resources in the group.
palTokens
[in, size_is(cResource)] Array of resources in the group.
plToken
[out] Returned group resource identifier.

Return Values

Returns an HRESULT value that depends on the implementation. **HRESULT** can be one of the following standard constants, or other values not listed:

Value	Meaning
E_FAIL	Failure.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method isn't supported.
S_OK or NOERROR	Success.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IResourceManager::ReleaseFocus

IResourceManager Interface

Sets the focus object to NULL in the resource manager if the current focus object is the one specified in this method.

HRESULT ReleaseFocus(
IUnknown* *pFocusObject*
);

Parameters

pFocusObject
[in] Focus object.

Return Values

Returns an HRESULT value that depends on the implementation. **HRESULT** can be one of the following standard constants, or other values not listed:

Value	Meaning
E_FAIL	Failure.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method isn't supported.
S_OK or NOERROR	Success.

Remarks

Use this method when the object of focus is about to be destroyed to ensure that the focus is not still being referenced.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IResourceManager::RequestResource

IResourceManager Interface

Requests the use of a given registered resource.

HRESULT RequestResource(
LONG *idResource*,
IUnknown* *pFocusObject*,
IResourceConsumer* *pConsumer*
);

Parameters

idResource
[in] Resource token retrieved when the resource was registered.
pFocusObject
[in] IUnknown interface of a focus object associated with a request (normally the filter's **IUnknown** interface).
pConsumer
[in] IResourceConsumer interface on the object requesting the resource.

Return Values

Returns an HRESULT value. Returns S_OK if the requested resource is returned, or S_FALSE if the resource is not available, in which case the resource manager will call the requesting object back when the resource becomes available. Any other return is an error.

Remarks

When there is more than one request for the resource, the resource manager will decide the priority by using the object of focus passed with each request and comparing it to the object of focus passed in the most recent IResourceManager::SetFocus method.

Requests will be filled in the following order of priority.

1. Requests made with exactly the same object of focus as the last SetFocus method.
2. Requests whose object of focus shares a common source filter.
3. Requests whose object of focus shares a common filter graph.
4. Requests in the same process as the focus.

While checking this priority, the resource manager will use QueryInterface on the focus object for IID_IFilter. If found, the resource manager will use IBaseFilter methods to check the filter graph and look for common source filters with the current focus object.

A filter should pass the IUnknown interface of the filter in the *pFocusObject* parameter. The filter graph manager matches filters to the filter graph and will attempt to trace filters to common source filters when checking objects of focus.

The focus object must be valid for the entire lifetime of the request — until either the IResourceManager::CancelRequest method is called, or the IResourceManager::NotifyRelease method is called with the *bStillWant* parameter set to FALSE.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IResourceManager::SetFocus

IResourceManager Interface

Notifies the resource manager that a specified object has been given the focus of the user.

HRESULT SetFocus(
IUnknown* *pFocusObject*
);

Parameters

pFocusObject
[in] Object that has been given the focus of the user.

Return Values

Returns an HRESULT value that depends on the implementation. **HRESULT** can be one of the following standard constants, or other values not listed:

Value	Meaning
E_FAIL	Failure.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method isn't supported.
S_OK or NOERROR	Success.

Remarks

In DirectShow, the object given the user's focus is typically a video renderer whose window has received the focus. The resource manager gives priority to requests for resources in the following order.

1. Requests made with the focus object specified in the *pFocusObject* parameter.
2. Requests whose focus object shares a common source filter.
3. Requests whose focus object shares a common filter graph.
4. Requests in the same process as the focus.

Once a focus has been set, the resource manager must maintain a focus object until ReleaseFocus is called. That is, after calling this method, you must use **ReleaseFocus** before the IUnknown interface of the focus object becomes invalid, unless you can guarantee that **SetFocus** is called by a different object in the meantime. No reference count is held on the focus object.

The resource manager will hold this pointer until replaced or canceled, and will use it to resolve resource contention. It will use QueryInterface for the IBaseFilter interface at least and, if found, will use methods on that interface. It calls methods on **IBaseFilter** to decide which audio renderer to use if there are two (it will choose the one with a source filter common to the focus object), and also to determine if the two objects are within the same filter graph.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

ISeekingPassThru Interface

The **ISeekingPassThru** interface is exposed on video renderer filters. It has only one method, Init, which you use to instantiate and initialize a CRendererPosPassThru object. Use this object to keep track of reference times and stream times. The IMediaSeeking and IMediaPosition interfaces can use these times to seek to various places in multimedia files.

When to Implement

Implement this interface when you write a video renderer filter that needs to keep track of reference time and stream time.

When to Use

Use this interface in your application when you want to create a CRendererPosPassThru class object.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Retrieves pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

ISeekingPassThru methods	Description
Init	Initializes a CRendererPosPassThru renderer-seeking object.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

ISeekingPassThru::Init

ISeekingPassThru Interface

Initializes a CRendererPosPassThru renderer-seeking object.

```
HRESULT Init(
  BOOL bSupportRendering,
  IPin *pPin
  );
```

Parameters

bSupportRendering
[in] TRUE indicates the pin specified in *pPin* is a renderer pin; FALSE indicates not a renderer pin.

pPin
[in] Pointer to the filter's input pin.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. Current DirectShow implementation return values include:

Value	Meaning
E_FAIL	Failed to create and initialize a CRendererPosPassThru object.
E_OUTOFMEMORY	Not enough memory to create the object.
NOERROR	Successfully created and initialized a CRendererPosPassThru object.

Remarks

This method instantiates and initializes a CRendererPosPassThru object.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IStandardCutList Interface

The **IStandardCutList** interface provides a simple way for an application to feed a cutlist into a cutlist provider (filter).

The IStandardCutList::AddElement method provides the primary functionality of this interface, by taking a pointer to a cutlist element and adding it to the list. The first clip added to a cutlist determines the media type. All other clips must be of the same media type. Removing clips from the cutlist is not supported.

The filter graph must be stopped when you call many of the methods on this interface.

See About Cutlists and Using Cutlists for more information.

When to Implement

Do not implement this interface. DirectShow implements it for you.

When to Use

Use this interface in your application when you need to create a whole cutlist out of individual cuts (elements).

When compiling a cutlist application you must explicitly include the cutlist header file as follows:

```
#include <cutlist.h>
```

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Retrieves pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IStandardCutList methods	Description
AddElement	Accepts a cutlist element from the application and adds it to the cutlist.
RemoveElement	Removes an element from a cutlist. (Not currently implemented.)
GetFirstElement	Retrieves the first element you added to cutlist.
GetLastElement	Retrieves the last element you added to cutlist.
GetNextElement	Retrieves the next element in the cutlist.
GetPreviousElement	Retrieves the previous element in the cutlist.
GetMediaType	Retrieves the clip's media type structure.
SetMediaType	Sets the media type for all clips in the cutlist.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IStandardCutList::AddElement

IStandardCutList Interface

Accepts a cutlist element from the application and adds it to the cutlist.

HRESULT AddElement(

IAMCutListElement **pElement*,
REFERENCE_TIME *mtStart*,
REFERENCE_TIME *mtDuration*
);

Parameters

pElement
[in] Pointer to the cutlist element to be added to the cutlist.
mtStart
[in] Relative position of the cut in the cutlist. Must be CL_DEFAULT_TIME (indicating that the relative position is the end of the current cutlist).
mtDuration
[in] Length of the cut. Must be CL_DEFAULT_TIME (indicating the duration is defined by the element).

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_INVALIDARG	Argument is invalid.
E_NOTIMPL	Method is not supported.
E_OUTOFMEMORY	Could not allocate the element descriptor.
S_OK	Success.

Remarks

This method adds a clip to the end of the clip list. The cutlist will play in the order you add the clips.

You can't call **AddElement** on this cutlist after you have given the cutlist to the graph builder by calling ICutListGraphBuilder::AddCutList. The **AddElement** call will be ignored. Make sure you have called **AddElement** as many times as you need to before calling **ICutListGraphBuilder::AddCutList**.

Removing clips from the cutlist is not supported.

The first clip added to a cutlist determines the media type. All other clips must be of the same media type.

See Also

IAMCutListElement

Previous Home Topic Contents Index Next

IStandardCutList::GetFirstElement

IStandardCutList Interface

Retrieves the first element you added to the cutlist.

HRESULT GetFirstElement(
IAMCutListElement ***ppElement*
);

Parameters

ppElement
[out] Address of a pointer to the first element in the cutlist.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_INVALIDARG	Argument is invalid.
E_NOTIMPL	Method is not supported.
E_OUTOFMEMORY	Could not allocate required memory.
S_OK	Success.

Remarks

You can only call this method when the graph is stopped. If you call this method while the graph is playing or paused, unpredictable behavior will result, including corrupting the cutlist that is playing.

This method increments the reference count on the cutlist element object. Be sure to decrement the cutlist element's reference count by calling its Release method as follows.

```
*ppElement->Release();
```

See Also

GetLastElement, IAMCutListElement

IStandardCutList::GetLastElement

IStandardCutList Interface

Retrieves the last element you added to the cutlist.

HRESULT GetLastElement(
IAMCutListElement ***ppElement*
);

Parameters

ppElement
[out] Address of a pointer to the last element you added to the cutlist.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_INVALIDARG	Argument is invalid.
E_NOTIMPL	Method is not supported.
E_OUTOFMEMORY	Could not allocate required memory.
S_OK	Success.

Remarks

You can only call this method when the graph is stopped. If you call this method while the graph is playing or paused, unpredictable behavior will result, including corrupting the cutlist that is playing.

This method increments the reference count on the cutlist element object. Be sure to decrement the cutlist element's reference count by calling its Release method as follows.

```
*ppElement->Release();
```

See Also

GetFirstElement, IAMCutListElement

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IStandardCutList::GetMediaType

IStandardCutList Interface

Retrieves the clip's media type structure.

HRESULT GetMediaType(
AM_MEDIA_TYPE *pmt*
);

Parameters

pmt
[in] Pointer to the AM_MEDIA_TYPE structure describing the clip.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_INVALIDARG	Argument is invalid.
E_NOTIMPL	Method is not supported.
E_OUTOFMEMORY	Could not allocate required memory.
S_OK	Success.

Remarks

This method retrieves the media type of all clips in the cutlist.

The first clip added to a cutlist determines the media type. All other clips must be of the same media type.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IStandardCutList::GetNextElement

IStandardCutList Interface

Retrieves the next element in the cutlist.

HRESULT GetNextElement(
IAMCutListElement ***ppElement*
);

Parameters

ppElement
[out] Address of a pointer to the next cutlist element.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure. You must call GetFirstElement or GetLastElement.
E_INVALIDARG	Argument is invalid.
E_NOTIMPL	Method is not supported.
E_OUTOFMEMORY	Could not allocate required memory.
S_FALSE	There is no next element.
S_OK	Success.

Remarks

You can only call this method when the graph is stopped. If you call this method while the graph is playing or paused, unpredictable behavior will result, including corrupting the cutlist that is playing.

You must call GetFirstElement or GetLastElement before this method will succeed.

This method increments the reference count on the cutlist element object. Be sure to decrement the cutlist element's reference count by calling its Release method as follows.

```
*ppElement->Release();
```

See Also

GetPreviousElement, IAMCutListElement

IStandardCutList::GetPreviousElement

IStandardCutList Interface

Retrieves the previous element in the cutlist.

HRESULT GetPreviousElement(
IAMCutListElement ***ppElement*
);

Parameters

ppElement
[out] Address of a pointer to the previous cutlist element.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure. You must call GetFirstElement or GetLastElement.
E_INVALIDARG	Argument is invalid.
E_NOTIMPL	Method is not supported.
E_OUTOFMEMORY	Could not allocate required memory.
S_FALSE	There is no previous element.
S_OK	Success.

Remarks

You can only call this method when the graph is stopped. If you call this method while the graph is playing or paused, unpredictable behavior will result, including corrupting the cutlist that is playing.

You must call GetFirstElement or GetLastElement before this method will succeed.

This method increments the reference count on the cutlist element object. Be sure to decrement the cutlist element's reference count by calling its Release method as follows.

```
*ppElement->Release();
```

See Also

GetNextElement, IAMCutListElement

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IStandardCutList::RemoveElement

IStandardCutList Interface

Removes an element from a cutlist. (Not currently implemented.)

HRESULT RemoveElement(
IAMCutListElement **pElement*
);

Parameters

pElement
[in] Pointer to the element to be removed.

Previous Home Topic Contents Index Next

IStandardCutList::SetMediaType

IStandardCutList Interface

Sets the media type for all clips in the cutlist.

HRESULT SetMediaType(
AM_MEDIA_TYPE **pmt*
);

Parameters

pmt
[in] Pointer to the AM_MEDIA_TYPE structure describing the clip.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_INVALIDARG	Argument is invalid.
E_UNEXPECTED	Too late to set the media type.
S_OK	Success.

Remarks

This method tells a cutlist what media type all of the elements in the cutlist must have. If you do not call this method, the first non-NULL element given to the cutlist through IStandardCutList::AddElement will determine the media type of the cutlist. All subsequent calls to AddElement must be of the same media type.

If you call this method, you must do so before ever calling AddElement. This method limits the elements that can be added to elements of the specified media type.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IUnknown Interface

The **IUnknown** interface lets clients get pointers to other interfaces on a given object through the IUnknown::QueryInterface method, and manage the existence of the object through the IUnknown::AddRef and IUnknown::Release methods. All other Component Object Model (COM) interfaces are inherited, directly or indirectly, from **IUnknown**. Therefore, the three methods in **IUnknown** are the first entries in the vtable for every interface.

Note that this interface and its methods are fully described in the COM documentation and are only partially documented here for quick reference.

When to Implement

You must implement IUnknown as part of every interface. If you are using C++ multiple inheritance to implement multiple interfaces, the various interfaces can share one implementation of **IUnknown**. If you are using nested classes to implement multiple interfaces, you must implement **IUnknown** once for each interface you implement.

Note that the IUnknown interface is implemented by the CUnknown base class in the DirectShow™ class library and so is inherited by most other classes.

When to Use

Use IUnknown methods to switch between interfaces on an object, add references, and release objects.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

Previous Home Topic Contents Index Next

IUnknown::AddRef

IUnknown Interface

Increments the reference count for the calling interface on an object. It should be called for every new copy of a pointer to an interface on a given object.

ULONG AddRef(void);

Return Values

Returns an integer from 1 to *n*, the value of the new reference count. This information is meant to be used for diagnostic/testing purposes only, because, in certain situations, the value might be unstable.

Previous Home Topic Contents Index Next

IUnknown::QueryInterface

IUnknown Interface

Returns a pointer to a specified interface on a component to which a client currently holds an interface pointer. This method must use IUnknown::AddRef on the pointer it returns.

HRESULT QueryInterface(
  REFIID *iid*,
  void ** *ppvObject*

);

Parameters

iid
[in] Specifies the IID of the interface being requested.
ppvObject
[out] Receives a pointer to an interface pointer to the object on return. If the interface specified in *iid* is not supported by the object, *ppvObject* is set to NULL.

Return Values

Returns S_OK if the interface is supported, S_FALSE if not.

Previous Home Topic Contents Index Next

IUnknown::Release

IUnknown Interface

Decrements the reference count for the calling interface on an object. If the reference count on the object falls to zero, the object is freed from memory.

ULONG Release(void);

Return Values

Returns the resulting value of the reference count, which is used for diagnostic/testing purposes only. If you need to know that resources have been freed, use an interface with higher-level semantics.

Previous Home Topic Contents Index Next

IVideoWindow Interface

The **IVideoWindow** interface supports the video window properties of a video renderer. It is a dual interface (accessible through Microsoft® Visual Basic® and Visual C++®) that controls a generic video window. Generally, this is a video renderer that draws video into a window on the display. The **IVideoWindow** interface supports both properties and methods. Properties are more easily accessible from many Automation controllers (such as Microsoft Visual Basic).

However, some operations require several properties to be changed simultaneously; for this reason, methods are provided that allow a number of related properties to be changed simultaneously. For example, setting the window's position and size can be done by four individual put_[property name] calls or by the single method SetWindowPosition.

The methods require only that the video renderer be connected. If it is not connected, all the interface functions return VFW_E_NOT_CONNECTED. Properties set on a video renderer persist between successive connections and disconnections. All applications should ensure that they reset the renderer properties before starting a presentation.

Because this interface is Automation-compatible, there are two important aspects to remember about parameters accepted by these methods. First, all Boolean returns are OAFALSE (0) or OATRUE (–1), which is different from the C or C++ definition. Second, all strings are defined as being of type BSTR. All strings sent to the interface should be allocated through the Automation SysAllocString function, and similarly all strings returned from the interface should be freed by using the Automation SysFreeString function.

When to Implement

The video renderer filter supplied with Microsoft DirectShow™ implements this interface. It is also implemented by the filter graph manager (via a plug-in distributor) to pass method calls from the application to the video renderer filter's implementation of the interface.

Implement this interface if you are writing a replacement video renderer filter. You can use the CBaseVideoWindow class, which handles the IDispatch implementation for Automation, to help implement this interface.

When to Use

This interface is used by applications or other filters that must control the video window's properties.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IDispatch methods	Description
GetTypeInfoCount	Determines whether there is type information available for this dispinterface.
GetTypeInfo	Retrieves the type information for this dispinterface if GetTypeInfoCount returned successfully.
GetIDsOfNames	Converts text names of properties and methods (including arguments) to their corresponding DISPIDs.
Invoke	Calls a method or accesses a property in this dispinterface if given a DISPID and any other necessary parameters.

IVideoWindow methods	Description
put_Caption	Sets the text caption on the playback window.
get_Caption	Retrieves the text caption on the playback window.
put_WindowStyle	Sets the playback window style.
get_WindowStyle	Retrieves the playback window style.
put_WindowStyleEx	Sets the style of the control window.
get_WindowStyleEx	Retrieves the playback window's extended style bits.
put_AutoShow	Specifies if the window will be automatically shown on the first state change.
get_AutoShow	Returns if the window will be automatically shown on the first state change.
put_WindowState	Sets the current window state (such as visible or minimized).
get_WindowState	Retrieves the current window state (such as visible or minimized).
put_BackgroundPalette	Informs the renderer to realize its palette in the background.
get_BackgroundPalette	Returns whenever the renderer realizes its palette in the background.
put_Visible	Sets the visibility of the window.
get_Visible	Retrieves the visibility of the window.
put_Left	Sets the x-axis coordinate for the video window.
get_Left	Retrieves the x-axis coordinate for the video window.
put_Width	Sets the width of the video window.
get_Width	Retrieves the width of the video window.
put_Top	Sets the y-axis coordinates for the video window.
get_Top	Retrieves the y-axis coordinates for the video window.
put_Height	Sets the height of the video window.
get_Height	Retrieves the height of the video window.
put_Owner	Sets the owning parent window for the video playback window.
get_Owner	Retrieves the owning parent window for the video playback window.
put_MessageDrain	Specifies a window to which the video window will post messages.
get_MessageDrain	Retrieves the window set to receive messages from the video window.
get_BorderColor	Retrieves the border color for the video window.
put_BorderColor	Sets the border color for the video window.
get_FullScreenMode	Returns the full-screen rendering mode of the video renderer filter supporting this interface.
put_FullScreenMode	Sets the full-screen mode for the video renderer filter supporting this interface.
SetWindowForeground	Tells the renderer filter to become the foreground window.
NotifyOwnerMessage	Forwards messages that have been received by a parent window to a child window owned by a filter.
SetWindowPosition	Sets the video window position on the display.
GetWindowPosition	Retrieves the video window position.
GetMinIdealImageSize	Retrieves the ideal minimum image size for the video image playback (client) area.
GetMaxIdealImageSize	Retrieves the ideal maximum image size for the video image playback (client) area.

GetRestorePosition	Returns the normal restored window dimensions.
HideCursor	Hides the cursor.
IsCursorHidden	Determines if the cursor is hidden or showing.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IVideoWindow::get_AutoShow

IVideoWindow Interface

Retrieves information about whether the window will be automatically shown.

HRESULT get_AutoShow(
long **AutoShow*
);

Parameters

AutoShow
[out] OATRUE indicates that the window will be made visible when the state is changed to the paused or running state.

Return Values

Returns an HRESULT value.

Previous Home Topic Contents Index Next

IVideoWindow::get_BackgroundPalette

IVideoWindow Interface

Retrieves information about whether any palette required will be realized in the background.

HRESULT get_BackgroundPalette(
long **pBackgroundPalette*

);

Parameters

pBackgroundPalette
[out] OATRUE indicates that the palette will be realized in the background.

Return Values

Returns an HRESULT value.

Previous Home Topic Contents Index Next

IVideoWindow::get_BorderColor

IVideoWindow Interface

Retrieves the border color for the video window.

HRESULT get_BorderColor(
long **pColor*
);

Parameters

pColor
[out] Retrieved border color as a COLORREF value.

Return Values

Returns an HRESULT value.

Previous Home Topic Contents Index Next

IVideoWindow::get_Caption

IVideoWindow Interface

Retrieves the textual title string for the video window.

HRESULT get_Caption(
BSTR **strCaption*
);

Parameters

strCaption
[out] Retrieved window title caption.

Return Values

Returns an HRESULT value.

IVideoWindow::get_FullScreenMode

IVideoWindow Interface

Returns the full-screen rendering capabilities of the renderer filter supporting this interface.

HRESULT get_FullScreenMode(
long **FullScreenMode*
);

Parameters

FullScreenMode
[out] OATRUE if supporting full-screen video, or OAFALSE if not.

Return Values

Returns an HRESULT value.

Remarks

This method is called by the filter graph manager when asked to render the video to full-screen size. If the renderer does not have inherent support for full-screen playback, it should return E_NOTIMPL. Otherwise, it should return NOERROR. If the renderer does support full-screen playback, this method determines if it is currently switched on or off.

IVideoWindow::get_Height

IVideoWindow Interface

Sets the height of the video window.

HRESULT get_Height(
long *pHeight*
);

Parameters

pHeight
[out] Retrieved vertical dimension of the video window.

Return Values

Returns an HRESULT value.

Previous Home Topic Contents Index Next

IVideoWindow::get_Left

IVideoWindow Interface

Retrieves the x-axis coordinate for the video window.

HRESULT get_Left(
long *pLeft*
);

Parameters

pLeft
[out] The x-axis coordinate to be retrieved.

Return Values

Returns an HRESULT value.

Previous Home Topic Contents Index Next

IVideoWindow::GetMaxIdealImageSize

IVideoWindow Interface

Retrieves the ideal maximum image size for the video image playback (client) area.

HRESULT GetMaxIdealImageSize(
long *pWidth*,
long *pHeight*
);

Parameters

pWidth
[out] Image width.
pHeight
[out] Image height.

Return Values

Returns an HRESULT value.

Previous Home Topic Contents Index Next

IVideoWindow::get_MessageDrain

IVideoWindow Interface

Retrieves the window set to receive messages from the video window.

HRESULT get_MessageDrain(
OAHWND *Drain*
);

Parameters

Drain
[in] Window currently assigned to receive messages from the video window.

Return Values

Returns an HRESULT value.

Remarks

The IVideoWindow::put_MessageDrain description contains a list of the Microsoft Win32® messages passed to the window that is specified as a message drain.

Previous Home Topic Contents Index Next

IVideoWindow::GetMinIdealImageSize

IVideoWindow Interface

Retrieves the ideal minimum image size for the video image playback (client) area.

```
HRESULT GetMinIdealImageSize(
  long *pWidth,
  long *pHeight
  );
```

Parameters

pWidth
[out] Image width.
pHeight
[out] Image height.

Return Values

Returns an HRESULT value.

Previous Home Topic Contents Index Next

IVideoWindow::get_Owner

IVideoWindow Interface

Retrieves the owning parent for the video window.

HRESULT get_Owner(
OAHWND * *pOwner*
);

Parameters

pOwner
[out] Retrieved window handle.

Return Values

Returns an HRESULT value.

Previous Home Topic Contents Index Next

IVideoWindow::GetRestorePosition

IVideoWindow Interface

Returns the normal restored window dimensions.

HRESULT GetRestorePosition(
long **pLeft*,
long **pTop*,
long **pWidth*,
long **pHeight*
);

Parameters

pLeft
[out] Left x-axis coordinate of the window.
pTop
[out] Top y-axis coordinate of the window.
pWidth
[out] Width of the window in pixels.
pHeight
[out] Height of the window in pixels.

Return Values

Returns an HRESULT value.

Remarks

When the window is maximized or minimized, the window position methods return the actual window size. This method returns the dimensions that the window would be when restored. It is useful for applications that want to save a window state while the window is maximized or minimized.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IVideoWindow::get_Top

IVideoWindow Interface

Retrieves the y-axis coordinate of the video window.

```
HRESULT get_Top(
  long *pTop
  );
```

Parameters

pTop
[out] The y-axis origin to be retrieved.

Return Values

Returns an HRESULT value.

Previous Home Topic Contents Index Next

IVideoWindow::get_Visible

IVideoWindow Interface

Retrieves the visibility of the video window.

HRESULT get_Visible(
long *pVisible*
);

Parameters

pVisible
[out] OATRUE if the window is shown; otherwise, the window is hidden.

Return Values

Returns an HRESULT value.

Previous Home Topic Contents Index Next

IVideoWindow::get_Width

IVideoWindow Interface

Retrieves the width of the video window.

HRESULT get_Width(
long *pWidth*
);

Parameters

pWidth
[out] Width to be retrieved.

Return Values

Returns an HRESULT value.

Previous Home Topic Contents Index Next

IVideoWindow::GetWindowPosition

IVideoWindow Interface

Retrieves the current window rectangle (not the client rectangle) in device coordinates.

HRESULT GetWindowPosition(
long *pLeft*,
long *pTop*,
long *pWidth*,
long *pHeight*
);

Parameters

pLeft
[out] The x-axis origin of the window.
pTop
[out] The y-axis origin of the window.
pWidth
[out] Width of the window in pixels.
pHeight
[out] Height of the window in pixels.

Return Values

Returns an HRESULT value.

Remarks

This method has the same effect as individually calling the IVideoWindow::get_Left, IVideoWindow::get_Top, IVideoWindow::get_Width, and IVideoWindow::get_Height methods.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IVideoWindow::get_WindowState

IVideoWindow Interface

Returns the state of the video window.

HRESULT get_WindowState(
long *WindowState*
);

Parameters

WindowState
[out] Flags indicating the state of the video window.

Return Values

Returns an HRESULT value.

Remarks

This method retrieves a subset of the properties of the window state, specifically SW_MINIMIZE, SW_MAXIMIZE, SW_SHOW, or SW_HIDE. These have the same definitions as the Microsoft Win32 ShowWindow function.

Previous Home Topic Contents Index Next

IVideoWindow::get_WindowStyle

IVideoWindow Interface

Changes the style parameters for the video window.

HRESULT get_WindowStyle(
long **pWindowStyle*
);

Parameters

pWindowStyle
[out] Set of flags that matches a subset of the flags that can be set by the GWL_STYLE value of the Microsoft Win32 GetWindowLong function.

Return Values

Returns an HRESULT value.

Remarks

For a complete list of window styles, see the CreateWindow function in the Microsoft Win32 Software Development Kit (SDK).

Previous Home Topic Contents Index Next

IVideoWindow::get_WindowStyleEx

IVideoWindow Interface

Changes the style parameters for the video window.

HRESULT get_WindowStyleEx(
long * *pWindowStyleEx*
);

Parameters

pWindowStyleEx
[out] Set of flags that matches a subset of the flags that can be set by the GWL_STYLE value of the Microsoft Win32 GetWindowLong function.

Return Values

Returns an HRESULT value.

Remarks

This function uses extended window styles. For a complete list of window styles, see the CreateWindow function in the Microsoft Win32 Software Development Kit (SDK).

Previous Home Topic Contents Index Next

IVideoWindow::HideCursor

IVideoWindow Interface

Hides the cursor.

HRESULT HideCursor(
long *HideCursor*
);

Parameters

HideCursor
[in] If OATRUE, do not display the cursor; if OAFALSE, display the cursor.

Return Values

Returns an HRESULT value.

Remarks

This method is typically used when the video renderer is in full-screen mode, where cursor display might be unwanted.

Previous Home Topic Contents Index Next

IVideoWindow::IsCursorHidden

IVideoWindow Interface

Determines if the cursor is hidden or showing.

```
HRESULT IsCursorHidden(
  long * CursorHidden
  );
```

Parameters

CursorHidden
[out] If OATRUE, cursor is hidden; if OAFALSE, cursor is displayed.

Return Values

Returns an HRESULT value.

Remarks

This method is typically used when the video renderer is in full-screen mode, where cursor display might be unwanted.

Previous Home Topic Contents Index Next

IVideoWindow::NotifyOwnerMessage

IVideoWindow Interface

Forwards messages that have been received by a parent window to a child window owned by a filter.

HRESULT NotifyOwnerMessage(
long *hwnd*,
long *uMsg*,
long *wParam*,
long *lParam*
);

Parameters

hwnd
[in] Window handle.
uMsg
[in] Message being sent.
wParam
[in] Message's *wParam* passed in.
lParam
[in] Message's *lParam* passed in.

Return Values

Returns an HRESULT value.

Remarks

This method should be used by windows that make a renderer window a child window. It forwards significant messages to the child window that the child window would not otherwise receive. This includes the following messages.
WM_ACTIVATEAPP
WM_DEVMODECHANGE
WM_DISPLAYCHANGE
WM_PALETTECHANGED
WM_PALETTEISCHANGING
WM_QUERYNEWPALETTE
WM_SYSCOLORCHANGE

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IVideoWindow::put_AutoShow

IVideoWindow Interface

Determines whether or not the window will be automatically shown.

HRESULT put_AutoShow(
long *AutoShow*
);

Parameters

AutoShow
[in] OATRUE (–1) means the window will be visible when the state changes; OAFALSE (0) means the window remains hidden until explicitly shown.

Return Values

Returns an HRESULT value.

Remarks

Many simple applications require a displayed window when a filter graph is set to the running state. *AutoShow* defaults to OATRUE so that when the graph changes state to paused or running, the window is visible (it also is set as the foreground window). It will remain visible on all subsequent state changes to paused or running. If you close the window while the stream is running, the window will not automatically reappear. If you stop and restart the stream, however, the window will automatically reappear.

See Also

IVideoWindow::put_Visible

Previous Home Topic Contents Index Next

IVideoWindow::put_BackgroundPalette

IVideoWindow Interface

Determines whether any palette required will be realized in the background.

HRESULT put_BackgroundPalette(
long *BackgroundPalette*
);

Parameters

BackgroundPalette
[in] OATRUE to realize the palette in the background; otherwise, OAFALSE.

Return Values

Returns an HRESULT value.

Remarks

If this is OATRUE (–1), any palette required by the video is realized by the renderer in the background. This means that any colors the palette uses will change to their closest match in the display palette prior to drawing. This ensures that an application will not have its palette disturbed when playing a video. It does, however, impose severe performance penalties on the video and should not be used unless absolutely necessary. The default value for this property is OAFALSE.

Previous Home Topic Contents Index Next

IVideoWindow::put_BorderColor

IVideoWindow Interface

Sets the border color for the video window.

HRESULT put_BorderColor(
long *Color*
);

Parameters

Color
[in] New border color as a COLORREF type.

Return Values

Returns an HRESULT value.

Remarks

When a destination rectangle that is set differs from the visible client area of the window, a border is exposed around the edge. This method allows an application to change the border color. It is set to black by default. Any nonsystem color passed in is converted to its closest match according to the current palette before being used (this is not an issue on true color

devices). Setting this causes the window border to be repainted in the new color automatically.

Previous Home Topic Contents Index Next

IVideoWindow::put_Caption

IVideoWindow Interface

Sets the textual title string for the video window.

HRESULT put_Caption(
BSTR *strCaption*
);

Parameters

strCaption
[in] Window title caption.

Return Values

Returns an HRESULT value.

Previous Home Topic Contents Index Next

IVideoWindow::put_FullScreenMode

IVideoWindow Interface

Sets the full-screen mode for the video renderer filter supporting this interface.

HRESULT put_FullScreenMode(
long *FullScreenMode*
);

Parameters

FullScreenMode
[in] OATRUE if supporting full-screen video, or OAFALSE if not.

Return Values

Returns E_NOTIMPL if the video renderer doesn't support full-screen mode or NOERROR if it does.

Remarks

This method allows an application to switch a full-screen renderer into and out of full-screen mode. The renderer's behavior when switched out of full-screen mode is implementation-dependent. The Microsoft full-screen renderer, for example, switches back to a window.

The IVideoWindow plug-in distributor in the filter graph manager implements full-screen renderer switching. It looks to see if any renderer in the graph supports a full-screen mode and, if not, will temporarily replace the renderer with the default DirectShow full-screen renderer. It calls IVideoWindow::GetMaxIdealImageSize to determine if a window can be made a topmost window and resized to the entire display. This is preferred to swapping renderers, because the filter graph might be using DirectDraw® overlays or a hardware decoder filter.

Previous Home Topic Contents Index Next

IVideoWindow::put_Height

IVideoWindow Interface

Sets the height of the video window.

HRESULT put_Height(
long *Height*
);

Parameters

Height
[in] New vertical dimension of the video window.

Return Values

Returns an HRESULT value.

Remarks

Calling this method does not affect the y-axis coordinate of the video window.

Previous Home Topic Contents Index Next

IVideoWindow::put_Left

IVideoWindow Interface

Sets the x-axis coordinate for the video window.

HRESULT put_Left(
long *Left*
);

Parameters

Left
[in] The x-axis coordinate to be set.

Return Values

Returns an HRESULT value.

Remarks

Calling this method does not affect the video window's width.

Previous Home Topic Contents Index Next

IVideoWindow::put_MessageDrain

IVideoWindow Interface

Specifies a window to which the video window will post messages.

HRESULT put_MessageDrain(
OAHWND *Drain*
);

Parameters

Drain

[in] Window to which messages will be posted.

Return Values

Returns an HRESULT value.

Remarks

The video renderer passes messages to the specified message drain by calling the Microsoft Win32 PostMessage function. These messages allow you to write applications that include user interaction, such as applications that require mouse clicks on specific areas of the video display. An application can have a close relationship with the video window and know at certain time points to look for user interaction. When the renderer passes a message to the drain, it sends the parameters, such as the client-area coordinates, exactly as generated.

DirectShow passes the following messages to the window specified by the *Drain* parameter, if and when the application generates them.

WM_KEYDOWN
WM_KEYUP
WM_LBUTTONDBLCLK
WM_LBUTTONDOWN
WM_LBUTTONUP
WM_MBUTTONDBLCLK
WM_MBUTTONDOWN
WM_MBUTTONUP
WM_MOUSEACTIVATE
WM_MOUSEMOVE
WM_NCLBUTTONDBLCLK
WM_NCLBUTTONDOWN
WM_NCLBUTTONUP
WM_NCMBUTTONDBLCLK
WM_NCMBUTTONDOWN
WM_NCMBUTTONUP
WM_NCMOUSEMOVE
WM_NCRBUTTONDBLCLK
WM_NCRBUTTONDOWN
WM_NCRBUTTONUP
WM_RBUTTONDBLCLK
WM_RBUTTONDOWN
WM_RBUTTONUP

Because this member function does not make the message drain window a child window, applications with full-screen capabilities can use it.

Previous Home Topic Contents Index Next

IVideoWindow::put_Owner

IVideoWindow Interface

Sets an owning parent for the video window.

HRESULT put_Owner(
OAHWND *Owner*
);

Parameters

Owner
[in] Handle of new owner window.

Return Values

Returns an HRESULT value.

Remarks

This method offers a way for applications to set the owner of the video window. This is often used when playing videos in compound documents. This method changes the parent of the renderer window and sets the WS_CHILD style for the video window.

To forward video window messages to the parent window, use the IVideoWindow::put_MessageDrain method, supplying the window handle of the parent window. This method does not post messages automatically.

After using this method to set the owner of a video window, you must reset the owner to NULL (by calling put_Owner(NULL)) before releasing the filter graph. Otherwise, messages will continue to be sent to this window and errors will likely occur when the application is terminated.

Previous Home Topic Contents Index Next

IVideoWindow::put_Top

IVideoWindow Interface

Sets the y-axis coordinate of the video window.

HRESULT put_Top(
long *Top*
);

Parameters

Top
[in] The y-axis origin of the video window.

Return Values

Returns an HRESULT value.

Remarks

Calling this method does not affect the height of the video window.

Previous Home Topic Contents Index Next

IVideoWindow::put_Visible

IVideoWindow Interface

Changes the visibility of the video window.

HRESULT put_Visible(
long *Visible*
);

Parameters

Visible
[in] Boolean flag that is compatible with Automation.

Return Values

Returns an HRESULT value.

Remarks

If the *Visible* parameter is set to OATRUE (–1), the window is shown. If it is set to OAFALSE (0), the window is hidden.

IVideoWindow::put_Width

IVideoWindow Interface

Sets the video window's width.

HRESULT put_Width(
long *Width*
);

Parameters

Width
[in] Width to be set.

Return Values

Returns an HRESULT value.

Remarks

Calling this method does not affect the video window's x-axis coordinate.

IVideoWindow::put_WindowState

IVideoWindow Interface

Sets the video window's state.

HRESULT put_WindowState(
long *WindowState*
);

Parameters

WindowState
[in] Describes the video window's state.

Return Values

Returns NOERROR.

Remarks

This method is a wrapper for the Microsoft Win32 ShowWindow function. **IVideoWindow::put_WindowState** passes the *WindowState* parameter on to CBaseWindow::DoShowWindow, which in turn passes it on to **ShowWindow**. Hence, *WindowState* can be any value that is valid for **ShowWindow**.

Previous Home Topic Contents Index Next

IVideoWindow::put_WindowStyle

IVideoWindow Interface

Changes the style parameters for the video window.

HRESULT put_WindowStyle(
long *WindowStyle*
);

Parameters

WindowStyle
[in] Set of flags that matches a subset of the flags that can be set by the GWL_STYLE value of the Microsoft Win32 GetWindowLong function.

Return Values

Returns an HRESULT value.

Remarks

Use this property to change the overall style of the video window; for example, to remove the border and caption areas of the video window. It is a fairly thin wrapper on top of setting the GWL_STYLE value of the Microsoft Win32 GetWindowLong function and therefore must be treated with care. In particular, ensure that the current styles are first retrieved, and then the necessary bit fields are added or removed. With some exceptions (noted here), the acceptable flags are the same as those allowed by the Win32 CreateWindow function.

Do not use this method to affect the window size. For example, if the window is minimized, do not set the WS_MAXIMIZE style; doing so causes unpredictable results. Instead, use the IVideoWindow::put_WindowState method for maximizing or minimizing the window.

Any of the following styles return E_INVALIDARG.
WS_DISABLED
WS_HSCROLL
WS_ICONIC
WS_MAXIMIZE
WS_MINIMIZE
WS_VSCROLL

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IVideoWindow::put_WindowStyleEx

IVideoWindow Interface

Sets the style of the control window.

HRESULT put_WindowStyleEx(
long *pWindowStyleEx*
);

Parameters

pWindowStyleEx
[in] Value that specifies the style of the control window.

Return Values

Returns NOERROR.

Remarks

This method uses EX window styles. For a complete list of extended window styles, see the CreateWindowEx function in the Microsoft Win32 Software Development Kit (SDK).

Use this property to change the overall style of the video window; for example, to remove the border and caption areas of the video window. It is a fairly thin wrapper on top of setting the GWL_STYLE value of the Microsoft Win32 GetWindowLong function and therefore must be

treated with care. In particular, ensure that the current styles are first retrieved, and then the necessary bit fields are added or removed.

Note: Do not use the following window styles as they are not validated.
WS_DISABLED
WS_HSCROLL
WS_ICONIC
WS_MAXIMIZE
WS_MINIMIZE
WS_VSCROLL

With some exceptions (noted here), the acceptable flags are the same as those allowed by the Win32 CreateWindow function.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IVideoWindow::SetWindowForeground

IVideoWindow Interface

Moves the video window to the foreground and optionally gives it focus.

HRESULT SetWindowForeground(
long *Focus*
);

Parameters

Focus
Long value that specifies whether the video window will get focus. A value of –1 gives the window focus and 0 does not.

Return Values

Returns one of the following values.

Value	Meaning
NOERROR	The method succeeded.
E_INVALIDARG	*Focus* doesn't equal –1 or 0.
VFW_E_NOT_CONNECTED	The current filter isn't connected to a complete filter graph.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IVideoWindow::SetWindowPosition

IVideoWindow Interface

Sets the position of the video window (not the client rectangle position) in device coordinates.

HRESULT SetWindowPosition(
long *Left*,
long *Top*,
long *Width*,
long *Height*
);

Parameters

Left
[in] The x-axis origin of the window.
Top
[in] The y-axis origin of the window.
Width
[in] Width of the window.
Height
[in] Height of the window.

Return Values

Returns an HRESULT value.

Remarks

This method has the same effect as individually calling the IVideoWindow::put_Left, IVideoWindow::put_Top, IVideoWindow::put_Width, and IVideoWindow::put_Height methods.

Specify, in window coordinates, where the video should appear. For example, setting a destination of (100,50,200,400) positions the video playback at an origin of 100 pixels from the left of the client area and 50 pixels from the top, with an overall size of 200 x 400 pixels. If the video is smaller than this (or a source rectangle has been specified that is smaller than the video), it will be stretched appropriately. Similarly, if the video is larger than the destination rectangle, the video is compressed into the visible rectangle. There are fairly severe performance penalties if an application does not keep the source and destination rectangles the same size.

Under typical circumstances, when no destination rectangle has been set, the video fills the

entire visible client window area (regardless of how much the user has stretched the window). Also, the destination rectangle properties correctly return the size of the video window client area.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IVPBaseConfig Interface

IVPBaseConfig enables a video port (VP) or overlay mixer filter to communicate with a VP driver (decoder), to set and retrieve configuration information. This interface assumes that the mixer filter creates the video port. The IVPConfig interface derives from this interface. See also IVPBaseNotify and IVPNotify.

When to Implement

The Windows Driver Model (WDM) Ksproxy filter implements this interface so you won't need to implement it in most cases. Implement this interface when you need this functionality on a platform that does not support WDM, or when you need to alter the default behavior.

When to Use

The Overlay Mixer filter uses this interface so you won't need to use it in most cases. Use this interface when you implement your own overlay mixer filter.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Retrieves pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IVPBaseConfig methods	Description
GetConnectInfo	Retrieves connection information structures.
SetConnectInfo	Sets the index for the current video port connection information.
GetVPDataInfo	Retrieves the current video port data information.
GetMaxPixelRate	Retrieves the maximum pixels per second rate for a given width and height.
InformVPInputFormats	Informs the device what video formats the video port supports.
GetVideoFormats	Retrieves the video formats the decoder supports.
SetVideoFormat	Sets the format that the video will use.
SetInvertPolarity	Reverses the current polarity the decoder uses.
GetOverlaySurface	Determines whether the overlay mixer should use the driver's overlay surface and if so retrieves a pointer to the surface.

SetDirectDrawKernelHandle	Sets the DirectDraw® kernel handle for the decoder's minidriver to use.
SetVideoPortID	Sets the port ID that the video will use.
SetDDSurfaceKernelHandle	Sets the kernel handle that the DirectDraw surface will use.
SetSurfaceParameters	Tells the capture driver about the surface created on its behalf by the Overlay Mixer or VBI surface filter.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IVPBaseConfig::GetConnectInfo

IVPBaseConfig Interface

Retrieves connection information structures.

HRESULT GetConnectInfo(
LPDWORD *pdwNumConnectInfo*,
LPDDVIDEOPORTCONNECT *pddVPConnectInfo*
);

Parameters

pdwNumConnectInfo
[in/out] Points to a buffer that contains the number of DDVIDEOPORTCONNECT structures provided by the *pddVPConnectInfo* parameter. Contains the actual number of structures returned on output. If *pddVPConnectInfo* is NULL, this method updates this parameter with the number of structures supported by the driver.
pddVPConnectInfo
[in/out] Points to an array of DDVIDEOPORTCONNECT structures that the driver fills in. Specify NULL to retrieve the total number of formats supported.

Return Values

Returns NOERROR if the count or structures were retrieved, or a driver error.

Remarks

This method retrieves the various connection information structures such as **GUID** and port width structures, in an array of structures specified by *pddVPConnectInfo*. The callee must allocate the correct amount of space for the number of structures requested.

Set the index for connection information by using the IVPBaseConfig::SetConnectInfo method.

IVPBaseConfig::GetMaxPixelRate

IVPBaseConfig Interface

Retrieves the maximum pixels per second rate for a given width and height.

```
HRESULT GetMaxPixelRate(
  LPAMVPSIZE pamvpSize,
  LPDWORD pdwMaxPixelsPerSecond
  );
```

Parameters

pamvpSize
[in/out] Pointer to an AMVPSIZE structure containing the desired width and height. The structure receives the final dimensions upon return.

pdwMaxPixelsPerSecond
[out] Pointer to the retrieved maximum pixels per second rate.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_POINTER	NULL pointer argument.
E_INVALIDARG	Invalid argument.
NOERROR	The maximum pixel rate was retrieved.

Remarks

This method retrieves the maximum pixels per second rate expected for a given format and a given scaling factor. If the decoder does not support those scaling factors, then it returns the rate and the nearest scaling factors it supports.

Previous Home Topic Contents Index Next

IVPBaseConfig::GetOverlaySurface

IVPBaseConfig Interface

Determines whether the overlay mixer should use the driver's overlay surface and if so retrieves a pointer to the surface.

HRESULT GetOverlaySurface(
LPDIRECTDRAWSURFACE **ppddOverlaySurface*
);

Parameters

ppddOverlaySurface
[out] Address of a pointer to the retrieved DirectDraw® overlay surface object.

Return Values

Returns NOERROR if the overlay surface object was returned. (Default implementation sets the surface to NULL and returns NOERROR.)

Remarks

The Overlay Mixer uses this function to determine if the driver requires the Overlay Mixer to use its overlay surface and if so to get a pointer to it. If this function returns NULL, then the Overlay Mixer allocates its own surface.

Previous Home Topic Contents Index Next

IVPBaseConfig::GetVideoFormats

IVPBaseConfig Interface

Retrieves the video formats the decoder supports.

HRESULT GetVideoFormats(
LPDWORD *pdwNumFormats*,
LPDDPIXELFORMAT *pddPixelFormats*
);

Parameters

pdwNumFormats
[in/out] Pointer to the number of DDPIXELFORMAT structures provided by the *pddPixelFormats* parameter. When called, this method updates this parameter with the actual number of structures retrieved. If *pddPixelFormats* is NULL, this method updates this parameter with the total number of formats the driver supports.
pddPixelFormats
[in/out] Pointer to an array of DDPIXELFORMAT structures that the driver fills. Specify NULL to retrieve only the count of supported formats in *pdwNumFormats*.

Return Values

Returns NOERROR if the count or structures were returned, or a driver error otherwise.

Remarks

This method queries for either the number of DDPIXELFORMAT structures supported by the driver, or retrieves as many structures as can fit into the provided buffer space.

The callee must allocate the correct amount of space for the number of structures requested.

Set the video format by using IVPBaseConfig::SetVideoFormat.

Previous Home Topic Contents Index Next

IVPBaseConfig::GetVPDataInfo

IVPBaseConfig Interface

Retrieves the current video port data information.

HRESULT GetVPDataInfo(
LPAMVPDATAINFO *pamvpDataInfo*
);

Parameters

pamvpDataInfo
[in/out] Pointer to the AMVPDATAINFO data information structure.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_POINTER	NULL pointer argument.
E_INVALIDARG	Invalid argument.
NOERROR	The video port data information was retrieved.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IVPBaseConfig::InformVPInputFormats

IVPBaseConfig Interface

Informs the device what video formats the video port supports.

HRESULT InformVPInputFormats(
DWORD *dwNumFormats*,
LPDDPIXELFORMAT *pDDPixelFormats*
);

Parameters

dwNumFormats
[in] Number of video formats contained in the *pDDPixelFormats* parameter.
pDDPixelFormats
[in] Array of pixel format structures (DDPIXELFORMAT) to send to the device.

Return Values

Returns S_FALSE if failure, or NOERROR otherwise.

Remarks

The supplied array of supported video port formats might determine what formats the device, in turn, proposes.

Previous Home Topic Contents Index Next

IVPBaseConfig::SetConnectInfo

IVPBaseConfig Interface

Sets the index for the current video port connection information.

HRESULT SetConnectInfo(
DWORD *dwChosenEntry*
);

Parameters

dwChosenEntry
[in] Index of new video port connect information (zero-based) to pass to the driver.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_INVALIDARG	Invalid argument.
NOERROR	The video port connect information was set.

Remarks

Retrieve connection information by using IVPBaseConfig::GetConnectInfo.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IVPBaseConfig::SetDDSurfaceKernelHandle

IVPBaseConfig Interface

Sets the kernel handle to be used by the DirectDraw surface.

HRESULT SetDDSurfaceKernelHandle(
DWORD *dwDDKernelHandle*
);

Parameters

dwDDKernelHandle
[in] DirectDraw surface handle for kernel mode, passed as a DWORD value.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_INVALIDARG	Invalid argument.
NOERROR	The specified handle is set successfully.

Remarks

This method sets the DirectDraw handle on the mini driver to enable it to communicate with the video port directly.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IVPBaseConfig::SetDirectDrawKernelHandle

IVPBaseConfig Interface

Sets the DirectDraw® kernel handle for the decoder's minidriver to use.

HRESULT SetDirectDrawKernelHandle(
DWORD *dwDDKernelHandle*
);

Parameters

dwDDKernelHandle
[in] DirectDraw kernel level handle passed as a DWORD value.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_INVALIDARG	Invalid argument.
NOERROR	The specified handle was set successfully.

Remarks

Sets the DirectDraw kernel level handle on the minidriver to enable it to communicate with DirectDraw directly.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IVPBaseConfig::SetInvertPolarity

IVPBaseConfig Interface

Reverses the current polarity the decoder uses.

HRESULT SetInvertPolarity(void);

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
NOERROR	The polarity was reversed.

Remarks

Reversing polarity means asking the decoder to treat even fields like odd fields and vice versa.

Previous Home Topic Contents Index Next

IVPBaseConfig::SetSurfaceParameters

IVPBaseConfig Interface

Tells the capture driver about the surface created on its behalf by the Overlay Mixer or VBI surface filter.

HRESULT SetSurfaceParameters(
DWORD *dwPitch*,
DWORD *dwXOrigin*,
DWORD *dwYOrigin* **) PURE;**

Parameters

dwPitch
[in] Pitch of the surface. Distance (or pitch) in pixels between the start pixels of two consecutive lines of the surface.
dwXOrigin
[in] X-value of the pixel at which valid data starts.
dwYOrigin
[in] Y-value of the pixel at which valid data starts.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. The current default implementation returns NOERROR if the call completed successfully, or E_NOTIMPL if the method is not implemented.

Previous Home Topic Contents Index Next

IVPBaseConfig::SetVideoFormat

IVPBaseConfig Interface

Sets the format to be used by the video.

HRESULT SetVideoFormat(
DWORD *dwChosenEntry*
);

Parameters

dwChosenEntry
[in] Specifies the index (zero-based) of the video pixel format to use.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_INVALIDARG	Invalid argument.
NOERROR	The new video format was set.

Remarks

Retrieve the video formats by using IVPBaseConfig::GetVideoFormats.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IVPBaseConfig::SetVideoPortID

IVPBaseConfig Interface

Sets the port ID which the video will use.

HRESULT SetVideoPortID (
DWORD *dwVideoPortID*
);

Parameters

dwVideoPortID
[in] DirectDraw video port ID.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_INVALIDARG	Invalid argument.
NOERROR	The specified port ID is set successfully.

Remarks

This method sets the DirectDraw video port ID on the mini driver to enable it to communicate with the video port directly.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IVPBaseNotify Interface

IVPBaseNotify enables you to control the properties of a filter that uses a video port. The IVPNotify interface derives from this interface. See also IVPBaseConfig and IVPConfig.

When to Implement

The Overlay Mixer filter implements this interface so you won't need to implement it in most cases. Implement this interface when you need to alter the default behavior.

When to Use

Use this interface in your application when you need to access video port properties.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Retrieves pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IVPBaseNotify methods	Description
RenegotiateVPParameters	Initializes the connection to the decoder.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IVPBaseNotify::RenegotiateVPParameters

IVPBaseNotify Interface

Initializes the connection to the decoder.

HRESULT RenegotiateVPParameters(void) PURE;

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_POINTER	NULL pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method is not supported.
NOERROR	No error.

Remarks

The Overlay Mixer filter negotiates various parameters (by using the IVPBaseConfig interface) with the decoder or driver. Call this function if any of those parameters (such as the video format or size) change. Currently, the Overlay Mixer repeats the whole connection process. You can call this method even while the graph is playing.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IVPConfig Interface

IVPConfig enables a video port (VP) or overlay mixer filter to communicate with a VP driver (decoder), to set and retrieve configuration information. This interface assumes that the mixer filter creates the video port. This interface derives from IVPBaseConfig. See also IVPBaseNotify and IVPNotify.

When to Implement

The Windows Driver Model (WDM) Ksproxy filter implements this interface so you won't need to implement it in most cases. Implement this interface when you need this functionality on a platform that does not support WDM, or when you need to alter the default behavior.

When to Use

The Overlay Mixer filter uses this interface so you won't need to use it in most cases. Use this interface when you implement your own overlay mixer filter.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Retrieves pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IVPBaseConfig methods	Description
GetConnectInfo	Retrieves connection information structures.
SetConnectInfo	Sets the index for the current video port connection information.
GetVPDataInfo	Retrieves the current video port data information.
GetMaxPixelRate	Retrieves the maximum pixels per second rate for a given width and height.
InformVPInputFormats	Informs the device what video formats the video port supports.
GetVideoFormats	Retrieves the video formats the decoder supports.
SetVideoFormat	Sets the format that the video will use.
SetInvertPolarity	Reverses the current polarity the decoder uses.
GetOverlaySurface	Determines whether the overlay mixer should use the driver's overlay surface and if so retrieves a pointer to the surface.
SetDirectDrawKernelHandle	Sets the DirectDraw® kernel handle for the decoder's minidriver to use.
SetVideoPortID	Sets the port ID that the video will use.
SetDDSurfaceKernelHandle	Sets the kernel handle that the DirectDraw surface will use.
SetSurfaceParameters	Tells the capture driver about the surface created on its behalf by the Overlay Mixer or VBI surface filter.

IVPConfig methods	Description
IsVPDecimationAllowed	Given the context, retrieves whether scaling at the video port is possible.
SetScalingFactors	Sets the factors by which the decoder should scale the video stream.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IVPConfig::IsVPDecimationAllowed

IVPConfig Interface

Given the context, retrieves whether scaling at the video port is possible.

HRESULT IsVPDecimationAllowed(
AMVP_CONTEXT *amvpContext*,
LPBOOL *pbIsDecimationAllowed*
);

Parameters

amvpContext
[in] Context (video or VBI) in which to query the VP decimation capability.
pbIsDecimationAllowed
[out] Pointer to the retrieved value indicating whether decimation is allowed.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_POINTER	NULL pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method is not supported.
NOERROR	No error.

Remarks

The Overlay Mixer filter uses this function to determine whether the driver needs the mixer to decimate video data at its own discretion. This function can be especially useful in a capture with preview situation in which you would not want the VP mixer filter to perform any scaling at the video port.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IVPConfig::SetScalingFactors

IVPConfig Interface

Sets the factors by which the decoder should scale the video stream.

HRESULT SetScalingFactors(
LPAMVPSIZE *pamvpSize*
);

Parameters

pamvpSize
[in] Pointer to the new scaling size structure (AMVPSIZE) to use to specify the width and

height.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
NOERROR	The new scaling factors were set.

Remarks

If the decoder does not support the specified scaling factors, then it sets the values to the nearest factors it can support.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IVPNotify Interface

IVPNotify enables you to control the properties of a filter that uses a video port. This interface derives from the IVPBaseNotify interface. See also IVPBaseConfig and IVPConfig.

When to Implement

The Overlay Mixer filter implements this interface so you won't need to implement it in most cases. Implement this interface when you need to alter the default behavior.

When to Use

Use this interface in your application when you need to access video port properties.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Retrieves pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IVPBaseNotify methods	Description
RenegotiateVPParameters	Initializes the connection to the decoder.

IVPNotify methods	Description
SetDeinterlaceMode	Sets the deinterlacing mode (such as bob or weave).
GetDeinterlaceMode	Retrieves the deinterlacing mode (such as bob or weave).
SetColorControls	Sets the color control settings associated with the specified overlay or primary surface.
GetColorControls	Retrieves the current color control settings associated with the specified overlay or primary surface.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IVPNotify::GetColorControls

IVPNotify Interface

Retrieves the current color control settings associated with the specified overlay or primary surface.

HRESULT GetColorControls(
LPDDCOLORCONTROL **ppColorControl*
) PURE;

Parameters

ppColorControl
[out] Address of the DDCOLORCONTROL structure that will receive the current control settings of the specified surface. The **dwFlags** member of the **DDCOLORCONTROL** structure indicates which of the color control options are supported.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

In the current DirectShow implementation, this method returns NOERROR if successful, or E_INVALIDARG or E_FAIL upon failure.

Previous Home Topic Contents Index Next

IVPNotify::GetDeinterlaceMode

IVPNotify Interface

Retrieves the mode (such as bob or weave).

HRESULT GetDeinterlaceMode(
AMVP_MODE **pmode*
) PURE;

Parameters

pmode
[out] Pointer to the retrieved mode. This value is a member of the AMVP_MODE enumerated data type.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

The current DirectShow implementation returns NOERROR for success or E_INVALIDARG if the argument is not valid.

Remarks

This method is not currently implemented and returns E_NOTIMPL.

Previous Home Topic Contents Index Next

IVPNotify::SetColorControls

IVPNotify Interface

Sets the color control settings associated with the specified overlay or primary surface.

HRESULT SetColorControls(
LPDDCOLORCONTROL *pColorControl*
) PURE;

Parameters

pColorControl
[in] Address of the DDCOLORCONTROL structure containing the new values to be applied to the specified surface.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

In the current DirectShow implementation, this method returns NOERROR if successful, or E_INVALIDARG or E_FAIL upon failure.

Previous Home Topic Contents Index Next

IVPNotify::SetDeinterlaceMode

IVPNotify Interface

Sets the mode (such as bob or weave).

HRESULT SetDeinterlaceMode(
AMVP_MODE *mode*
) PURE;

Parameters

mode
[in] Specified mode. This value is a member of the AMVP_MODE enumerated data type.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

The current DirectShow™ implementation returns NOERROR for success or E_INVALIDARG if *mode* is not a member of the AMVP_MODE enumerated data type.

Remarks

This method is not currently implemented and returns E_NOTIMPL.

DirectShow C++ Class Library

This section contains reference entries for all the DirectShow C++ classes, their data members, and their methods.

- Introduction to the DirectShow C++ Class Library
- CAggDirectDraw Class
- CAggDrawSurface Class
- CAMEvent Class
- CAMMsgEvent Class
- CAMSchedule Class
- CAMThread Class
- CAutoLock Class
- CBaseAllocator Class
- CBaseBasicVideo Class
- CBaseControlVideo Class
- CBaseControlWindow Class
- CBaseDispatch Class
- CBaseFilter Class
- CBaseInputPin Class
- CBaseList Class
- CBaseMediaFilter Class
- CBaseObject Class
- CBaseOutputPin Class
- CBasePin Class

- CBasePropertyPage Class
- CBaseReferenceClock Class
- CBaseRenderer Class
- CBaseStreamControl Class
- CBaseVideoRenderer Class
- CBaseVideoWindow Class
- CBaseWindow Class
- CBasicAudio Class
- CCmdQueue Class
- CCritSec Class
- CDeferredCommand Class
- CDisp Class
- CDispBasic
- CDispParams Class
- CDrawImage Class
- CEnumMediaTypes Class
- CEnumPins Class
- CFactoryTemplate Class
- CGenericList Class
- CGuidNameList Class
- CImageAllocator Class
- CImageDisplay Class
- CImagePalette Class
- CImageSample Class
- CLoadDirectDraw Class

- CMediaControl Class
- CMediaEvent Class
- CMediaPosition Class
- CMediaSample Class
- CMediaType Class
- CMemAllocator Class
- CMsg Class
- CMsgThread Class
- COARefTime Class
- COutputQueue Class
- CPersistStream Class
- CPosPassThru Class
- CPullPin Class
- CQueue Class
- CRefTime Class
- CRenderedInputPin Class
- CRendererInputPin Class
- CRendererPosPassThru Class
- CSource Class
- CSourcePosition Class
- CSourceSeeking Class
- CSourceStream Class
- CSystemClock Class
- CTransformFilter Class
- CTransformInputPin Class

- CTransformOutputPin Class
- CTransInPlaceFilter Class
- CTransInPlaceInputPin Class
- CTransInPlaceOutputPin Class
- CUnknown Class
- CVideoTransformFilter Class
- FOURCCMap Class

Previous Home Topic Contents Index Next

Introduction to the DirectShow C++ Class Library

This article provides a general description of the Microsoft® DirectShow™ class library, the relationship of the base classes to the DirectShow Component Object Model (COM) interfaces that they implement, and describes the utility classes that are not directly associated with interfaces. This article does not provide low-level descriptions of each class, nor does it provide specific instructions on how to use them to build a filter or run the filter graph manager.

The DirectShow C++ class library can help you implement the required interfaces on filters that you write. Most base classes correspond directly to interfaces, while other utility classes allow integration of Microsoft Win32® functionality, such as critical sections and thread management.

Contents of this article:

- Base Classes
  - CBaseObject and CUnknown Classes
  - Base Classes that Implement Interfaces
    - Filter Base Classes
    - Pin Base Classes
    - Enumerator Base Classes
    - Transport Base Classes
    - Media Control and Positioning Classes
    - Clock Base Classes
- Utility Classes
  - Win32 Classes
  - List and Queue Classes
  - Multimedia Data Type Classes
  - COM Classes
  - Debugging Classes

Base Classes

Most of the base classes in the DirectShow class library implement DirectShow COM interfaces. These classes produce C++ objects that provide an IUnknown interface so external components can access the interfaces the objects support.

CBaseObject and CUnknown Classes

The CBaseObject class is the root of all base classes. It exists primarily to provide debugging assistance by keeping a count of all DirectShow objects that are active. All derived base class constructors provide a debugging object name as the first parameter and call the **CBaseObject** constructor. You can view the debugging object name sent to this base class on

a debugging monitor.

All DirectShow classes that implement interfaces derive from a base class called CUnknown, which is derived from CBaseObject. **CUnknown** implements the INonDelegatingUnknown interface which, like the IUnknown interface, provides methods to request an interface, and to add or release references to that interface.

Why are there two interfaces that implement the services of IUnknown? Because of *aggregation*. Aggregation is the COM term for the combining of more than one object into a single larger object. Although filter graph objects, such as filters and pins, are rarely aggregated, the design is available for future extensibility and also for implementing plug-in distributors (PID), which are objects that are aggregated with the filter graph manager. In an aggregated object, the *outer object* (the one containing the other objects) uses the **IUnknown** interface to communicate outside the object. The **IUnknown** interface on the outer object passes out references to the **IUnknown** interfaces of its internal objects. That is, when an application calls the **IUnknown** interface on the outer object and asks for the interface belonging to one of its internal objects, the outer object calls the **IUnknown** interface of the internal object to retrieve the requested interface.

Because the internal objects must delegate IUnknown interfaces to the **IUnknown** of the outer object, the **IUnknown** interface of the internal object should not be accessed privately (that is, without going through the outer object's **IUnknown** interface). The internal object's **IUnknown** is reserved exclusively for communicating through the outer object. However, it is possible that objects will want to connect to other objects privately, without knowledge of the outer object. For example, pins on filters are likely to need to query interfaces on pins of other objects privately.

The INonDelegatingUnknown interface provides direct, private access to interfaces, regardless of whether or not the object is aggregated. Direct access is important in most of the communication between the DirectShow objects such as pins, allocators, and filters, and is the default method of communication. In fact, the base classes implement the IUnknown interface on nonaggregated objects (which includes almost every object in the filter graph) to call the nondelegating interface directly.

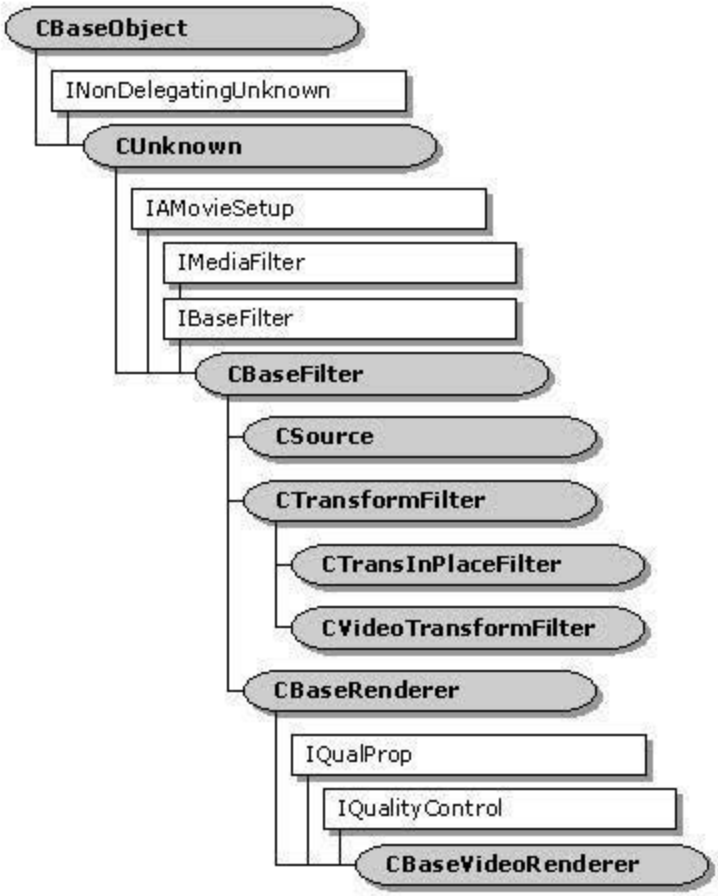
Base Classes that Implement Interfaces

The majority of classes in the DirectShow class library implement COM interfaces and can be categorized as follows:

- Filter base classes implement the IBaseFilter interface, and include CBaseFilter and classes derived from it.
- Pin classes implement the IPin interface, and include CBasePin and derived classes.
- Enumerator classes include CEnumPins and CEnumMediaTypes.
- Memory classes include CMediaSample, CBaseAllocator, CMemAllocator, and their derived classes.
- Control and position classes include CBaseFilter, CMediaPosition, CMediaControl, CBaseMediaFilter, CSourceSeeking, and CBaseStreamControl.

Filter Base Classes

The DirectShow stream architecture is based on filters and pins. Filters communicate with the filter graph manager and with the pins on the filter. Pins connect filters and handle transporting the data down the stream.



CBaseFilter is the base class for all filter classes. It implements the IBaseFilter interface, which specifies methods that allow the filter graph manager to create and enumerate pins, retrieve filter information, and notify the filter that it has been added to a filter graph. **CBaseFilter** also implements the IMediaFilter interface (from which **IBaseFilter** derives) to allow the filter to receive run, pause, and stop commands from the filter graph manager. This base class adds member functions to retrieve the pin count, retrieve pointers to individual pins, and retrieve the pin version.

The CBaseMediaFilter class also implements the IMediaFilter interface. However, because **IMediaFilter** is also implemented by CBaseFilter, this class is seldom used except to write a plug-in distributor (PID).

Several classes are derived directly from CBaseFilter. Each of these classes provides a base class for implementing a specific type of filter. These include:

- CSource, a base class for source filters.
- CTransformFilter, a base class for transform filters.
- CBaseRenderer, a base class for renderer filters.

The CSource filter class works in conjunction with the CSourceStream pin class to help create a source filter. Most of the work is done in the pin class, and **CSource** adds pin creation and deletion member functions. The CSourcePosition class implements a source filter.

The CTransformFilter class implements a transform filter. Derive your transform class from **CTransformFilter** if you want to make a copy of the data. The CTransInPlaceFilter class, derived from **CTransformFilter**, allows in-place transforms that do not copy the data. These transform filter classes work in conjunction with similarly named pin classes (for example, CTransformOutputPin and CTransformInputPin). Most member functions in the pin classes are implemented to call member functions in the transform filter class, so typically you need only to derive your filter from the filter class and override a few member functions to implement a transform filter.

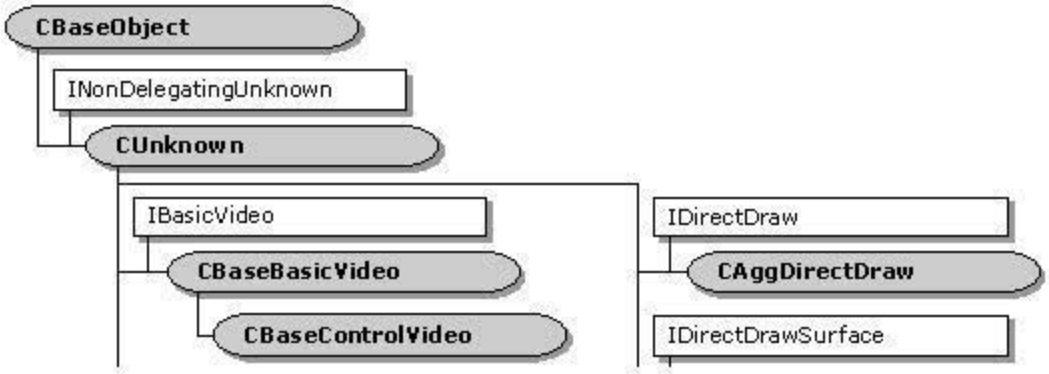
CTransformFilter adds several member functions to those inherited from CBaseFilter. Some of these are pure virtual member functions that the derived class must override. One example is the CTransformFilter::Transform member function, which is called when the input pin receives a sample. This member function provides the core of the transform functionality. Other member functions to be overridden also involve implementations that are specific to the derived class, such as verifying media types on pins and allocating the correct amount of memory. Additionally, several **CTransformFilter** member functions are called at various points in the connection or streaming process; the derived class can override these to handle requirements such as adding or releasing references to interfaces.

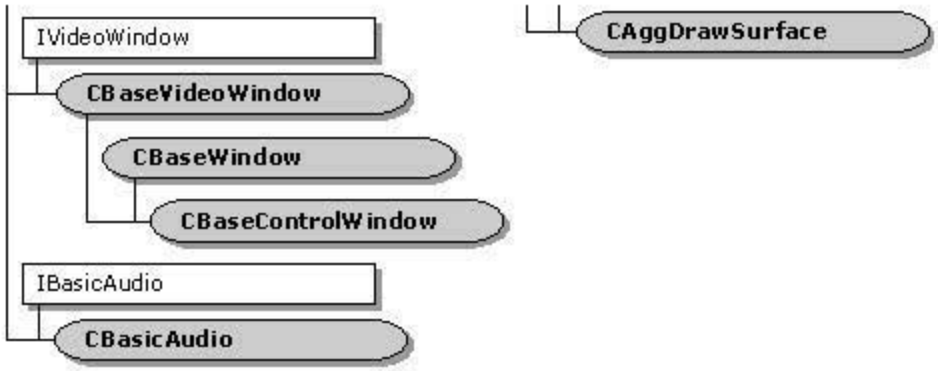
The CVideoTransformFilter class derives from the CTransformFilter class and is used as a base class for filters that can affect the quality of a rendered video by dropping frames when the video renderer sends quality-control messages. This class is primarily used by video decompressors in the DirectShow run time.

The CBaseRenderer class and its derived class, CBaseVideoRenderer, are the base filter classes that implement a video renderer filter. The video renderer filter used in DirectShow is derived from **CBaseVideoRenderer**. There are other renderer classes that work in conjunction with these classes but are not derived from CBaseFilter. These classes are:

- CRendererInputPin
- CBaseControlVideo and its base class CBaseBasicVideo
- CBaseControlWindow, and its base classes CBaseVideoWindow and CBaseWindow
- CAggDirectDraw
- CAggDrawSurface

The following illustration shows all the classes that support renderers that are not derived from either CBaseFilter or CBasePin.

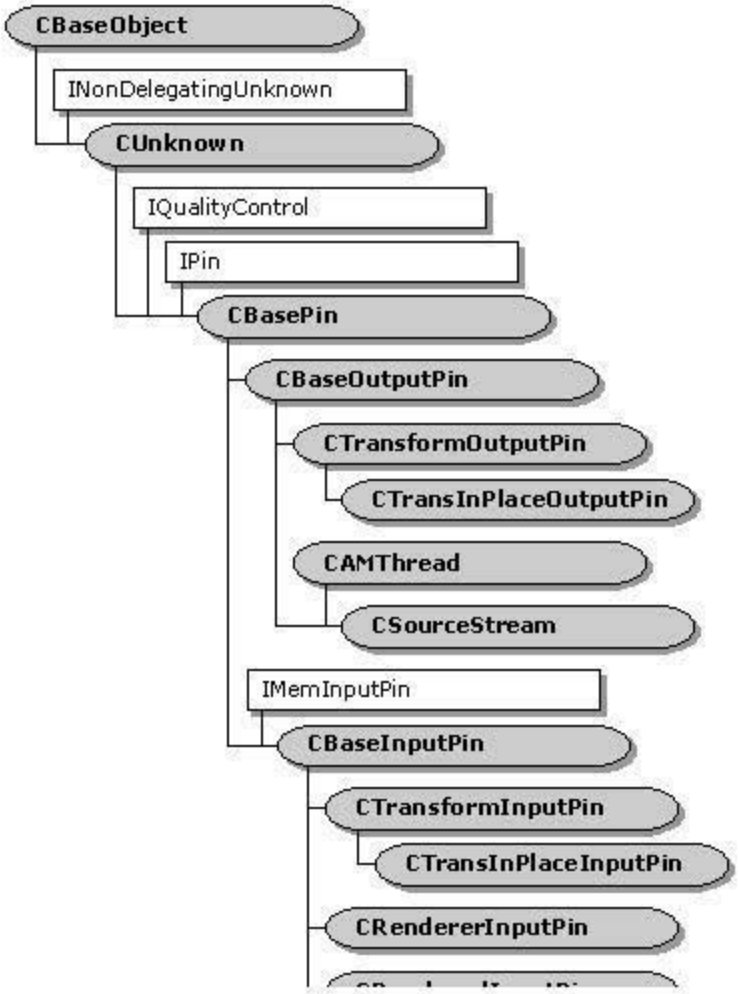




Pin Base Classes

Pins have a greater share of the work than filters. A pin must expose methods so that the filter graph manager can connect it with a pin on another filter. Pins also expose methods so that connected pins can negotiate what media type they will pass between them, and which pin will provide the shared memory allocator for transporting the media sample. Additionally, the output pin is responsible for passing each media sample to its connected input pin; the input pin is responsible for receiving it. Finally, pins must support interfaces so that quality-control messages and position information can be passed through the stream from pin to pin.

The following illustration shows the pin classes. All pin classes are derived from CBasePin, a base class derived from CUnknown.



CBasePin implements the IPin interface. The **IPin** interface specifies methods for connecting to other pins, negotiating the media type to be used with the connected pin, querying internal connections on the pin, and informing the pin of stream activity.

Besides implementing the IPin methods, CBasePin also implements IQualityControl methods so that quality-control messages can be passed through the filter graph from one pin to the next. Quality-control messages allow a filter, such as a renderer, to request another filter to adjust its sample rate. Typically, quality-control messages travel upstream from renderer to source filter. However, in cases such as a video capture filter, the source filter (for example, a VCR reader) can send quality-control messages downstream to the renderer filter to adjust its rate.

The CBasePin class provides several virtual member functions that can be overridden to provide handling of the connection, media type negotiation, and disconnection processes. Two base classes derive from **CBasePin** to provide default handling for many of these tasks:

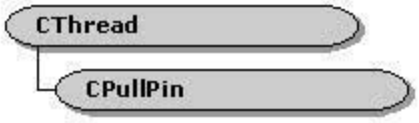
- CBaseOutputPin implements an output pin.
- CBaseInputPin implements an input pin.

CBaseOutputPin is the base class for the CTransformOutputPin and CSourceStream classes. Likewise, CBaseInputPin is the base class for the CTransformInputPin class. Before looking at these derived base pin classes, it is helpful to understand the basic model the **CBaseOutputPin** and **CBaseInputPin** classes use.

In the connection and transport model used by two pins, the input pin supports the IMemInputPin interface so that it can receive a media sample. The CBaseInputPin class implements the **IMemInputPin** interface. Also, one of the two pins must supply a shared memory allocator object, which is an object that contains the IMemAllocator interface that generates media sample objects passed between pins. An **IMemInputPin** method, implemented by the **CBaseInputPin** class, supplies this allocator object, implemented by the CMemAllocator class. The connected output pin also has the option of supplying its own allocator; if this is the case, it notifies the input pin (through another **IMemInputPin** method) of the final decision of which allocator is used.

The CBaseOutputPin class provides extra member functions to set the size and count of samples in the allocator, retrieve a media sample from the allocator, deliver that media sample to the connected input pin, and deliver end-of-stream and end-flush messages downstream. It also implements many of the IPin methods.

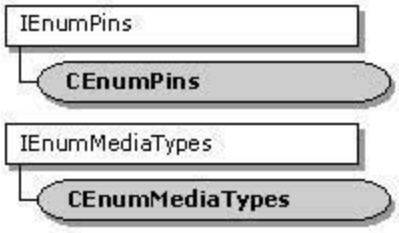
CPullPin is a class that is used on the input pin of a parser filter. It is derived from the CAMThread class as shown in the following illustration.



A *parser filter* pulls information from the disk, using the asynchronous file reader filter, or from the Internet, using the URL moniker filter. CPullPin works with the IAsyncReader interface, which is implemented on the source reader filter upstream. **CPullPin** starts the thread, pulls data from the upstream filter, and then pushes the data downstream. That is, it can simply call its own IMemInputPin::Receive method after pulling the sample from the source (or perform the equivalent routines elsewhere).

Enumerator Base Classes

An *enumerator* is an interface that provides methods for traversing a list of elements. Enumerators are used in COM programming, and the DirectShow model follows the COM model in enumerating objects. Two enumerator classes are provided in the class library: CEnumPins, which implements the IEnumPins interfaces, and CEnumMediaTypes, which implements the IEnumMediaTypes interface. Two other DirectShow enumerator interfaces, IEnumFilters and IEnumRegFilters, are not represented by base classes because they are implemented only by the filter graph manager.



The CEnumPins class creates an enumerator when the IBaseFilter::EnumPins method is called. The enumerator returned by this method is a pointer to the IEnumPins interface, which is implemented by the **CEnumPins** class. The **CEnumPins** member functions can then be called to retrieve pointers to each of the pins on the filter, which this enumerator accomplishes by calling the CBaseFilter::GetPin member function on the filter. The filter must override the base class **CBaseFilter::GetPin** member function to supply the enumerator with the next pin in the list each time it is called.

The CEnumMediaTypes class creates an enumerator when the IPin::EnumMediaTypes method is called. Pins store a list of the media types that they support. During negotiation of the media type, one pin typically calls the EnumMediaTypes method on its connected pin, retrieves the enumerator, and uses it to select a media type. Both of these enumerator classes support the **Next**, **Skip**, **Reset**, and **Clone** methods familiar to COM programmers. The media type enumerators call the CBasePin::GetMediaType member function, which must be overridden by the derived pin class, to return the next media type in a list of media types accepted by the pin.

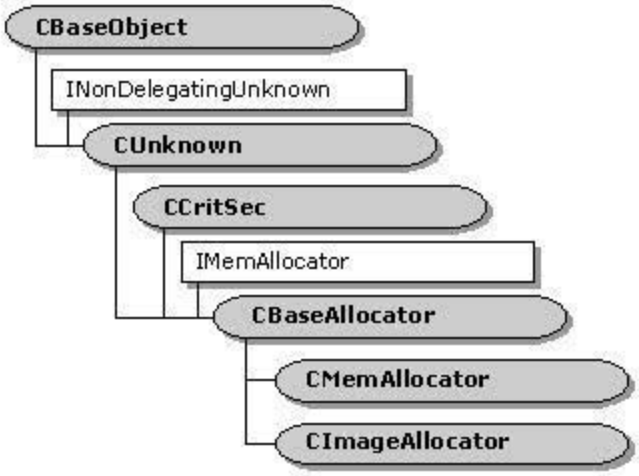
Enumerators operate as threads, and must have synchronized access to the pin media type list. For this reason, the classes that implement enumerators inherit (through multiple inheritance) from the CCritSec class, which provides critical section management. For more information about the **CCritSec** class, see Win32 Classes.

Transport Base Classes

Transport classes share memory between pins and pass media samples using that memory. DirectShow provides four classes to help implement shared memory transports:

- CBaseAllocator
- CMemAllocator
- CMediaSample
- CImageSample

CBaseAllocator is a class that provides member functions to implement the IMemAllocator interface, as shown in the following illustration.



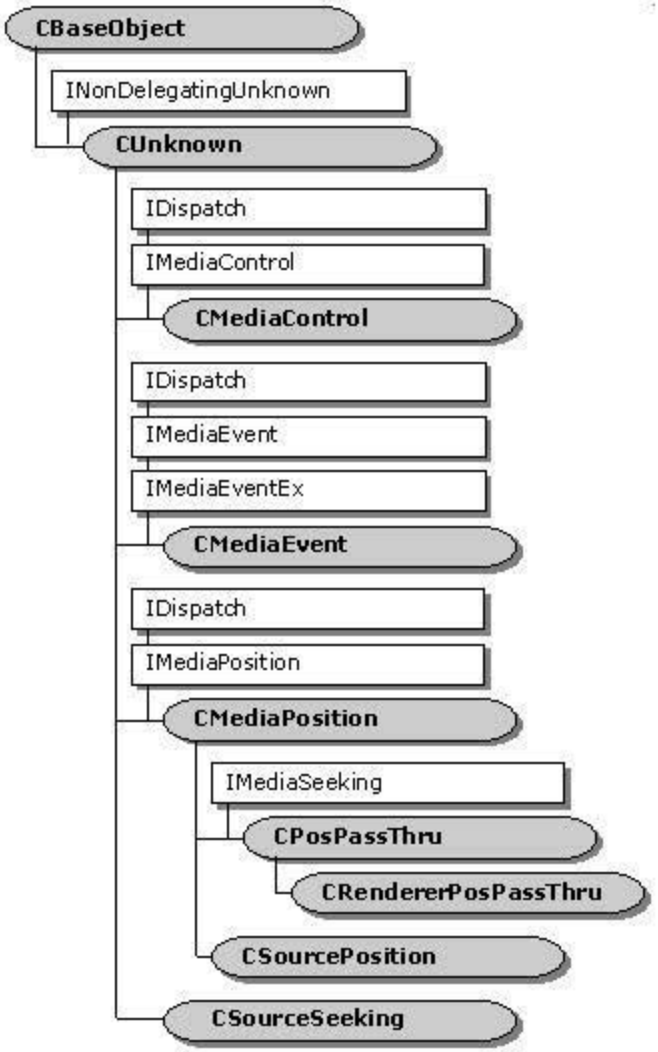
The IMemAllocator interface on the input pin specifies methods to set the number and size of the buffers to allocate, allocates that memory, frees that memory, and returns a single buffer that contains an IMediaSample interface. The output pin connected to the input pin calls the **IMemAllocator** methods. CBaseAllocator provides the member functions Alloc and Free that are called from the Commit and Decommit methods. Derived classes override the **Alloc** and **Free** member functions to provide their own routines to allocate and free memory.

Because CBaseAllocator performs very little implementation by itself, most pins use the CMemAllocator class, which is derived from **CBaseAllocator**. **CMemAllocator** overrides the CBaseAllocator::Free member function to provide allocation of media samples based on system memory. It provides its own member function, called ReallyFree, to be called when the allocator is finally released.

CMediaSample is a class that contains the media sample data and also provides member functions to access properties on the media sample, such as data type or beginning and ending time stamps. This class implements the IMediaSample interface, which provides the method specification. CImageSample derives from **CMediaSample** and is used by the video renderer when the renderer's allocator is being used. It uses all the **CMediaSample** interface methods and adds two methods to set and retrieve the DIBSECTION information. This makes it easy for the renderer to cast the **CMediaSample** pointer it receives from an upstream filter to a **CImageSample** pointer, and obtain a handle to the bitmap of the video frame.

Media Control and Positioning Classes

Media control interfaces pass commands such as **Run**, **Stop**, or **Pause** from an application through the filter graph manager to the individual filters. From the filter's perspective, the only control interface necessary is IMediaFilter, which exposes methods to accept and implement these commands. The CBaseFilter class implements this interface. All other interfaces that expose media control methods are handled by the filter graph manager and are therefore already implemented. Although a CMediaControl class exists and implements the IMediaControl interface, it is not often used because the filter graph manager is responsible for this functionality. The following illustration shows the relationship between these classes and interfaces.



Media positioning interfaces start the media stream at a specified position, play the stream for a specified period of time, or change the rate of the media stream. The IMediaPosition interface is the primary interface supporting this functionality. The CMediaPosition class implements this interface and serves as a base class for two other classes: CPosPassThru and CSourcePosition.

Typically, the filter graph manager calls the IMediaPosition interface on the renderer filters when it wants to position the media stream. The renderer acknowledges the sample times that it will be expected to display and then passes the media positioning data upstream, destined for a seekable filter, such as a source file filter, that can provide the properly positioned source stream. To pass that information upstream, output pins must be able to receive the positioning information.

The CPosPassThru class implements the IMediaPosition interface and the IMediaSeeking interface on the output pins of filters and, for the most part, does nothing but call the corresponding interface on the output pin of the next upstream filter, thereby passing through the positioning data. **IMediaSeeking** is different than **IMediaPosition** in that it allows the media stream to be seeked to units other than time, such as frames, samples, or indexed fields in an MPEG format. The CRendererPosPassThru class, implemented on a video renderer, sets the start and end reference times on individual samples, so that samples can be queried at any time for this information. This is helpful when dealing with seeking using **IMediaSeeking**, which seeks to *media time*, and does not keep track of the sample's reference time.

The reason for serially informing every filter in the graph of the new position is to allow filters that might be concerned with media positioning to be prepared for the new position. Certain stream splitters, for example, might be splitting off streams with media positions relative to the main media stream. This is why the filter graph manager does not simply call the source filter's IMediaPosition or IMediaSeeking interface directly.

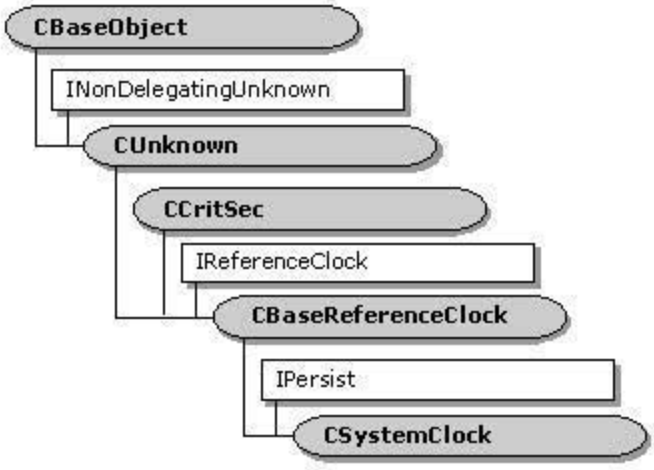
CSourcePosition is the class that helps the source filter implement its IMediaPosition interface.

The CSourceSeeking class helps the source filter implement its IMediaSeeking interface. This class enables a source filter to handle calls that change the start and stop positions in the media stream, and the playback rate.

The CBaseStreamControl class helps the source filter implement its IAMStreamControl interface. This class is used primarily by capture filters. The following illustration shows the relationship between **CBaseStreamControl** and the interfaces from which it inherits.

Clock Base Classes

DirectShow provides two classes, CBaseReferenceClock and CSystemClock to help implement clocks in the filter graph. The following illustration shows the relationship between these classes and the interfaces they implement.



CBaseReferenceClock implements IReferenceClock, and so provides the ability to return the correct reference time when requested, and to advise registered objects of specific times or time intervals through event notification and semaphores.

CSystemClock implements a system clock that provides time information and timing signals to an application. It uses the CBaseReferenceClock base class to provide most of that functionality, overriding the actual time calls.

Utility Classes

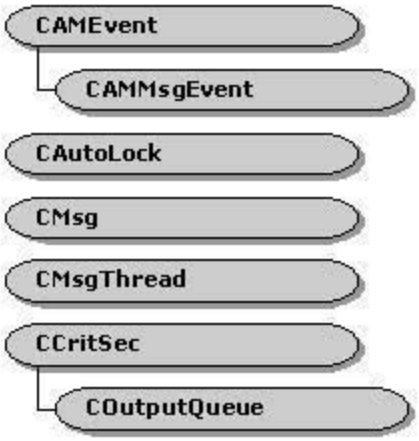
The DirectShow SDK includes several utility classes that provide C++ class encapsulation of many of the required Win32 functions, multimedia data structures, and object list and queue manipulation. These classes are briefly described in this section.

Win32 Classes

DirectShow implements several classes to handle Win32 threads, events, and critical sections. These include the following classes.

CAMEvent
CCritSec
CAutoLock
CAMThread
CMsgThread
CMsg

The following diagram illustrates these classes.



CAMEvent handles a Win32 event as a C++ object. The methods in this class allow events to be put into the signaled state or reset to a nonsignaled state, and also allow a caller to block until an event is signaled. Events can also be cast to handles and passed to the Win32 WaitForMultipleObjects function.

CCritSec handles a Win32 critical section as a C++ object to provide intraprocess synchronization. Methods of this class allow you to create, lock, and unlock a critical section.

CAutoLock holds a critical section (a CCritSec object) for the scope of a block or function. The critical section is locked in the constructor and unlocked in the destructor.

CAMThread provides an abstract worker thread class enabling creation, synchronization, and communication with a worker thread.

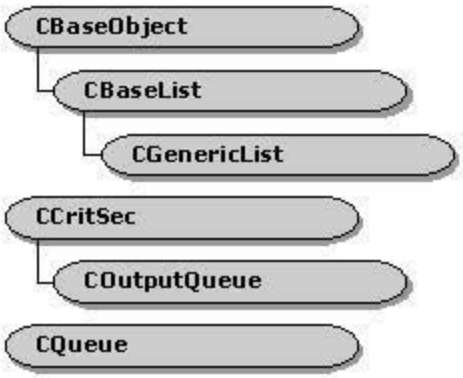
CMsgThread provides support for a worker thread to which requests can be posted

asynchronously instead of being sent directly. Messages, in the form of a CMsg object, can be posted to a **CMsgThread** object.

CMsg creates an object containing a message to be passed to a CMsgThread object.

List and Queue Classes

DirectShow implements the CBaseList, CGenericList, and COutputQueue classes for handling lists and queues as illustrated in the following diagram.



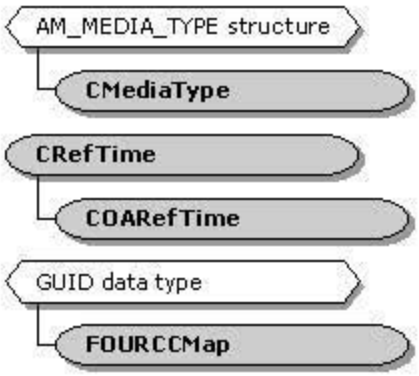
CBaseList represents a linked list data structure of typeless pointers to objects derived from CBaseObject.

CGenericList implements a template class derived from CBaseList that calls **CBaseList** member functions and adds type checking for the type specified in the template.

COutputQueue supports the queuing of media samples from the output pin of a filter. The output pin calls member functions of this class instead of calling methods on the connected input pin to receive the media sample. The output pin is then free to continue without blocking, while the **COutputQueue** class handles the passing of the media samples downstream.

Multimedia Data Type Classes

DirectShow implements the CMediaType, CRefTime, and FOURCCMap multimedia data type classes as shown in the following illustration.



CMediaType provides a C++ class object containing the media type data structure and methods that provide access to each of the members of the structure.

CRefTime provides a C++ class object containing the methods used to access the reference time, and operators used to perform Boolean tests or arithmetical operations on two **CRefTime** objects.

FOURCCMap provides conversion between the older-style FOURCC media tags used to identify and register media types and the GUID media subtypes used by DirectShow.

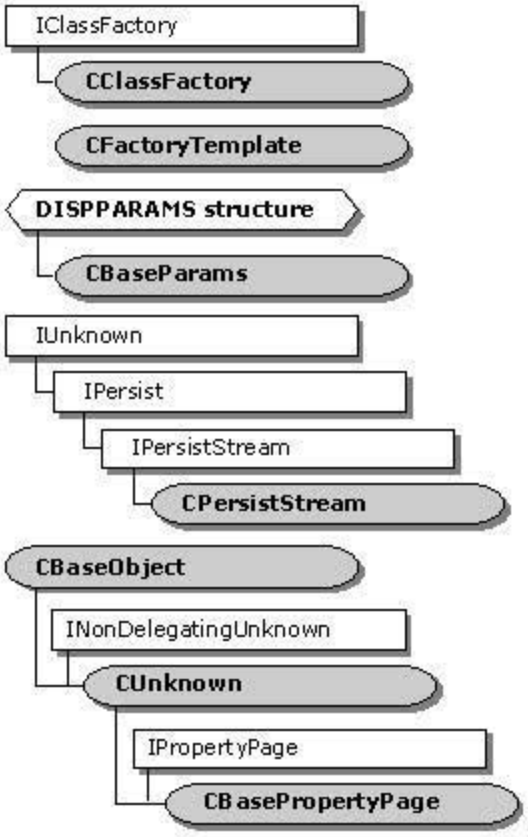
COM Classes

COM interface classes in DirectShow fall into two groups: object creation and interface implementation. Class factory classes are provided for object creation, and other classes are provided to implement existing COM interfaces.

The COM utility classes include the following.

CClassFactory
CFactoryTemplate
CPersistStream
CBasePropertyPage

The following illustration shows the relationship between the COM classes and the interfaces they implement.



CClassFactory and CFactoryTemplate are implemented by the base classes to handle automatic instantiation of filters, pins, and other DirectShow COM objects. These classes provide a scaffolding for object construction which wraps the actual COM elements required to construct an object. CPersistStream and CBasePropertyPage help with implementing COM persistent stream and property page interfaces.

with implementing COM persistent storage and property page interfaces.

CClassFactory, located in dllentry.cpp, inherits from CBaseObject and implements the COM IClassFactory interface. This interface is used by CoCreateInstance, which instantiates a COM object by calling IClassFactory::CreateInstance, which, in turn, calls the static **CreateInstance** member function in your derived class.

The base classes use CFactoryTemplate to provide CClassFactory with a template containing the CLSID of your object and a pointer to the static **CreateInstance** function for your object class.

CPersistStream implements COM IPersistStream for the storage and retrieval of filter properties in a saved filter graph. This enables a stored filter graph to have filters set to predefined property values. This class also provides a special member function to handle versioning of data in a stream.

CBasePropertyPage implements the COM IPropertyPage interface, which provides a framework for a property page associated with a filter.

Debugging Classes

DirectShow provides many debugging functions and macros as described in the Debugging reference section. It also includes three classes that aid in debugging filter development:

CDispBasic

CDisp

CGuidNameList

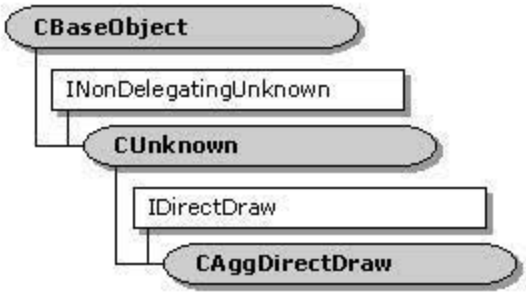
The following diagram illustrates these classes.

CDispBasic converts the m_PString data member to the proper string size.

CDisp provides a constructor that sets the CDispBasic class's m_PString data member to a string describing some relevant debugging information about the object used as a parameter to the constructor. For example, when constructed with an IPin pointer, **m_PString** returns the name of the pin; when constructed with a CLSID, **m_PString** returns a string representation of it, and so on. The class also provides an LPCTSTR cast operator that returns the value of **m_PString**, so the class can simply be cast as an LPCTSTR value to return the string when constructed.

CGuidNameList implements an array of globally unique identifier (GUID) names in the Uuids.h include file. This enables you to retrieve the **GUID** name for a media type, for example.

CAggDirectDraw Class



This class aggregates an IDirectDraw interface. Although DirectDraw® interfaces (**IDirectDraw** and IDirectDrawSurface) potentially have the ability to be aggregated, this feature is not yet implemented. Various parts of Microsoft® DirectShow™ require aggregation of the DirectDraw interfaces. In particular, the video renderer passes out media samples that expose **IDirectDraw** and **IDirectDrawSurface**. This class and the CAggDrawSurface class republish the methods of the DirectDraw class so that they can be aggregated.

Each member function in this class, with the exception of the constructor, SetDirectDraw, and NonDelegatingQueryInterface, simply calls the corresponding method on the IDirectDraw interface with the parameters passed to it.

Protected Data Members

Name	Description
m_pDirectDraw	DirectDraw object.

Member Functions

Name	Description
CAggDirectDraw	Constructs a CAggDirectDraw object.
SetDirectDraw	Sets the DirectDraw object to be aggregated by this class.

Overridable Member Functions

Name	Description
NonDelegatingQueryInterface	Returns an interface and increments the reference count.

Implemented IDirectDraw Methods

Name	Description
Compact	Moves all the pieces of surface memory on the video card to a contiguous block to make the largest chunk of free memory available.
CreateClipper	Creates a DirectDrawClipper object.
CreatePalette	Creates a DirectDrawPalette object for this DirectDraw object.
CreateSurface	Creates a DirectDrawSurface object for this DirectDraw object.
DuplicateSurface	Duplicates a DirectDrawSurface object.

EnumDisplayModes	Enumerates all the display modes the hardware exposes through the DirectDraw object that are compatible with a provided surface description.
EnumSurfaces	Enumerates all the existing or possible surfaces that meet the search criterion specified.
FlipToGDISurface	Makes the surface that GDI writes to the primary surface.
GetCaps	Fills in the raw (not remaining) capabilities of the device driver (the hardware) and/or the Hardware Emulation Layer (HEL).
GetDisplayMode	Returns the current display mode.
GetFourCCCodes	Gets the FOURCC codes supported by the DirectDraw object.
GetGDISurface	Returns the DirectDrawSurface object that currently represents the surface memory that GDI treats as the primary surface.
GetMonitorFrequency	Points to a DirectDrawSurface pointer that will be made to point to the DirectDrawSurface object currently controlling GDI's primary surface memory.
GetScanLine	Returns the scan line that the monitor is currently updating to the display.
GetVerticalBlankStatus	Returns the status of the vertical blank.
Initialize	Initializes the DirectDraw object.
RestoreDisplayMode	Resets the mode of the display device hardware for the primary surface to what it was before the CAggDirectDraw::SetDisplayMode member function was called.
SetCooperativeLevel	Determines the top-level behavior of the application.
SetDisplayMode	Sets the mode of the display device hardware.
WaitForVerticalBlank	Helps the caller synchronize itself with the vertical blank interval.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

CAggDirectDraw::CAggDirectDraw

CAggDirectDraw Class

Constructs a CAggDirectDraw object.

```
CAggDirectDraw(
  TCHAR *pName,
  LPUNKNOWN pUnk
  );
```

Parameters

pName
Name of the object; used for debugging purposes.
pUnk
Pointer to the owner of this object. If non-NULL, IUnknown calls are delegated to this object.

Return Values

No return value.

Remarks

This member function calls the CUnknown::CUnknown base class constructor and sets the m_pDirectDraw member variable to NULL.

Previous Home Topic Contents Index Next

CAggDirectDraw::NonDelegatingQueryInterface

CAggDirectDraw Class

Returns an interface and increments the reference count.

HRESULT NonDelegatingQueryInterface(
REFIID *riid*,
void ** *ppv*
);

Parameters

riid
Reference identifier.
ppv
Pointer to the interface.

Return Values

Returns E_POINTER if *ppv* is invalid. Returns NOERROR if the query is successful or E_NOINTERFACE if it is not.

Remarks

This member function provides an implementation of the INonDelegatingUnknown::NonDelegatingQueryInterface method. By default it passes out references to IDirectDraw and then calls the CUnknown::NonDelegatingQueryInterface

member function for base class interface references. Override this class to return interfaces added in the derived class.

Previous Home Topic Contents Index Next

CAggDirectDraw::SetDirectDraw

CAggDirectDraw Class

Sets the DirectDraw object to be aggregated by this class.

void SetDirectDraw(
LPDIRECTDRAW *pDirectDraw*
);

Parameters

pDirectDraw
IDirectDraw object to be aggregated.

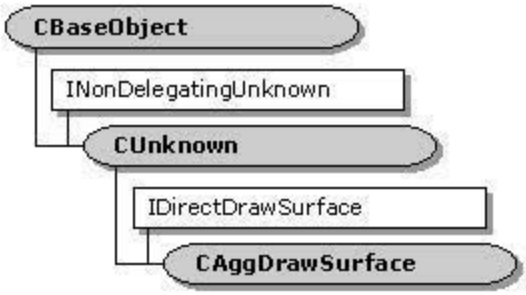
Return Values

No return value.

Remarks

This member function sets the m_pDirectDraw data member to the *pDirectDraw* parameter.

CAggDrawSurface Class



This class aggregates an IDirectDrawSurface interface. Although DirectDraw® interfaces (IDirectDraw and **IDirectDrawSurface**) potentially have the ability to be aggregated, this feature is not yet implemented. Various parts of Microsoft® DirectShow™ require aggregation of the DirectDraw interfaces. In particular, the video renderer passes out media samples that expose **IDirectDraw** and **IDirectDrawSurface**. This class and the CAggDirectDraw class republish the methods of the DirectDraw class so that they can be aggregated.

Each member function in this class, with the exception of the constructor, SetDirectDrawSurface, and NonDelegatingQueryInterface, simply calls the corresponding method on the IDirectDrawSurface interface with the parameters passed to it.

Protected Data Members

Name	Description
m_pDirectDrawSurface	DirectDraw surface.

Member Functions

Name	Description
CAggDrawSurface	Constructs a CAggDrawSurface object.
SetDirectDrawSurface	Sets the DirectDraw object to be aggregated by this class. This must be called before any of the IDirectDrawSurface interface methods can be called.

Overridable Member Functions

Name	Description
NonDelegatingQueryInterface	Returns an interface and increments the reference count.

Implemented IDirectDrawSurface Methods

Name	Description
AddAttachedSurface	Attaches a surface to another surface. Examples of possible attachments include z-buffers, alpha channels, and back buffers.
AddOverlayDirtyRect	Builds up the list of the rectangles that must be updated the next time the UpdateOverlayDisplay member function is called.
Blt	Performs a bit-block transfer.
BltBatch	Performs a sequence of CAggDrawSurface::Blt operations from several sources to a single destination.
BltFast	Performs a source copy bit-block transfer or transparent bit-block transfer using a source or destination color key.
DeleteAttachedSurface	Detaches two attached surfaces.
EnumAttachedSurfaces	Enumerates all the surfaces attached to a given surface.
EnumOverlayZOrders	Enumerates the overlays on the specified destination. The overlays can be enumerated in front-to-back or back-to-front order.
Flip	Makes the surface memory associated with the DDSCAPS_BACKBUFFER surface become associated with the FRONTBUFFER surface.
GetAttachedSurface	Finds the attached surface that has the specified capabilities.
GetBltStatus	Returns the status of a bit block transfer.
GetCaps	Returns the capabilities of the surface.
GetClipper	Returns the DirectDrawClipper object associated with this surface.
GetColorKey	Returns the color key value for the DirectDrawSurface object.
GetDC	Creates a GDI-compatible hDC for the surface.
GetFlipStatus	Returns OK if the surface that it is called on has finished its flipping process; otherwise, returns DDERR_WASSTILLDRAWING.
GetOverlayPosition	Returns the display coordinates of the surface, given a visible, active overlay surface (DDSCAPS_OVERLAY set).
GetPalette	Returns the DirectDrawPalette structure associated with this surface.
GetPixelFormat	Returns the color and pixel format of the surface.
GetSurfaceDesc	Returns a DDSURFACEDESC structure describing the surface in its current condition.
Initialize	Initializes a DirectDrawSurface object.
IsLost	Determines if the surface memory associated with a DirectDrawSurface object has been freed.
Lock	Obtains a valid pointer to the surface memory.
ReleaseDC	Releases a GDI-compatible hDC previously obtained through CAggDrawSurface::GetDC.
Restore	Restores a surface that has been "lost." The surface memory associated with the DirectDrawSurface object has been freed.
SetClipper	Attaches a DirectDrawClipper object to a DirectDrawSurface object.
SetColorKey	Sets the color key value for the DirectDrawSurface object if the hardware supports color keys on a per-surface basis.
SetOverlayPosition	Changes the display coordinates of an overlay surface.
SetPalette	Attaches the DirectDrawPalette object specified to a DirectDrawSurface.
Unlock	Notifies DirectDraw that the direct surface manipulations are complete.

UpdateOverlay	Repositions and/or modifies the visual attributes of an overlay surface. These surfaces must have the DDSCAPS_OVERLAY bit set.
UpdateOverlayDisplay	Repaints the rectangles in the dirty rectangle lists of all active overlays.
UpdateOverlayZOrder	Sets an overlay's z-order. The z-order determines which overlay should be occluded when multiple overlays are displayed simultaneously.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

CAggDrawSurface::CAggDrawSurface

CAggDrawSurface Class

Creates a CAggDrawSurface object.

CAggDrawSurface(
TCHAR *pName*,
LPUNKNOWN *pUnk*
);

Parameters

pName
Name of the object; used for debugging purposes.
pUnk
Pointer to the owner of this object. If non-NULL, IUnknown interface calls are delegated to this object.

Return Values

No return value.

Remarks

This member function calls the CUnknown::CUnknown base class constructor and sets the m_pDirectDrawSurface member variable to NULL.

Previous Home Topic Contents Index Next

CAggDrawSurface::NonDelegatingQueryInterface

CAggDrawSurface Class

Returns an interface and increments the reference count.

HRESULT NonDelegatingQueryInterface(
REFIID *riid*,
void ** *ppv*
);

Parameters

riid
Reference identifier.
ppv
Pointer to the interface.

Return Values

Returns E_POINTER if *ppv* is invalid. Returns NOERROR if the query is successful or E_NOINTERFACE if it is not.

Remarks

This member function provides an implementation of the INonDelegatingUnknown::NonDelegatingQueryInterface method. By default it passes out references to IDirectDrawSurface and then calls the CUnknown::NonDelegatingQueryInterface member function for base class interface references. Override this class to return interfaces added in the derived class.

Previous Home Topic Contents Index Next

CAggDrawSurface::SetDirectDrawSurface

CAggDrawSurface Class

Called by the owner of this aggregation object to set the actual DirectDraw surface it is aggregating upon.

void SetDirectDrawSurface(
LPDIRECTDRAWSURFACE *pDirectDrawSurface*
);

Parameters

pDirectDrawSurface
DirectDrawSurface to be set.

Return Values

No return value.

Remarks

This member function must be called before any of the IDirectDrawSurface interface methods can be called.

Previous Home Topic Contents Index Next

CAMEvent Class

CAMEvent

The **CAMEvent** class is an event object that can be set and waited on to provide interthread synchronization. This is currently implemented by using the Microsoft® Win32® **Event** application programming interfaces (APIs).

Events can be created as manual-reset or automatic-reset, and will always be created as not set (nonsignaled state). They can also be cast to handles so as to be passed to the Win32 WaitForMultipleObjects function.

Protected Data Members

Name	Description
m_hEvent	Microsoft Win32 event handle.

Member Functions

Name	Description
CAMEvent	Constructs a CAMEvent object.
Check	Returns TRUE if the event is currently set, but does not block.
Reset	Forces the event into a nonsignaled state.
Set	Puts the event into a signaled state.
Wait	Blocks until the event is signaled, or until an optional time-out occurs.
operator HANDLE	Gets the HANDLE object.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

CAMEvent::CAMEvent

CAMEvent Class

Constructs a CAMEvent object.

CAMEvent(

BOOL *fManualReset* **= FALSE**
);

Parameters

fManualReset
If this value is FALSE, the event is reset when the CAMEvent::Wait member function completes. If this parameter is TRUE, you can set the event by calling the CAMEvent::Set member function and then reset it by calling the CAMEvent::Reset member function.

Return Values

No return value.

Previous Home Topic Contents Index Next

CAMEvent::Check

CAMEvent Class

Returns TRUE if the event is currently set, but does not block.

BOOL Check(void);

Remarks

For events that are not manual-reset events, this member function causes the event to enter a nonsignaled state.

Previous Home Topic Contents Index Next

CAMEvent::Reset

CAMEvent Class

Forces the event into a nonsignaled state.

void Reset(void);

Return Values

No return value.

Previous Home Topic Contents Index Next

CAMEvent::Set

CAMEvent Class

Puts the event into a signaled state.

void Set(void);

Return Values

No return value.

Remarks

If the event is not a manual-reset event and there is at least one thread blocked on this event, the thread is released and the event remains in a nonsignaled state. If the event is not a manual-reset event and no threads are blocked on the event, it is set to a signaled state.

If the event is not a manual-reset event, it is set to a signaled state and all the threads blocked on this event are released.

Previous Home Topic Contents Index Next

CAMEvent::Wait

CAMEvent Class

Blocks until the event is signaled, or until the indicated time-out occurs.

BOOL Wait(
  DWORD *dwTimeout*
  );

Parameters

dwTimeout
Optional time-out value, represented in milliseconds. The default is INFINITE.

Return Values

Returns TRUE if the event becomes signaled; otherwise, returns FALSE.

Remarks

For events that are not manual-reset events, the action completing the **CAMEvent::Wait** member function causes the event to enter a nonsignaled state until the CAMEvent::Set member function is called.

Previous Home Topic Contents Index Next

CAMEvent::operator HANDLE

CAMEvent Class

Gets the HANDLE object associated with this CAMEvent object.

```
operator HANDLE () const;
```

Return Values

Returns the Microsoft Win32 event HANDLE.

‹Previous Home Topic Contents Index Next›

CAMMsgEvent Class

The **CAMMsgEvent** class is a wrapper for event objects that do message processing. This class adds one method to the CAMEvent object to allow sent messages to be processed while waiting.

Member Functions

Name	Description
WaitMsg	Allows sent messages to be processed while waiting for an event to be signaled or for the indicated time-out to occur.

‹Previous Home Topic Contents Index Next›

CAMMsgEvent::WaitMsg

CAMMsgEvent Class

Allows sent messages to be processed while waiting for an event to be signaled or for the indicated time-out to occur.

BOOL WaitMsg(
  DWORD *dwTimeOut*
  );

Parameters

dwTimeOut
Optional time-out value, represented in milliseconds. The default is INFINITE.

Return Values

Returns TRUE if the event is signaled, or FALSE if the time-out occurred.

Remarks

Call **CAMMsgEvent::WaitMsg** rather than CAMEvent::Wait if you want to block on a time-out or a signaled event and continue to process sent messages. If you do not process messages and another thread sends you a message, deadlock could occur. For example, if you create a thread by way of the Win32 CreateThread function and then block until the thread can initialize, deadlock will occur if the thread sends a message to your window using the Win32 SendMessage function. This is because **SendMessage** does not return until the message has been processed. **CAMMsgEvent::WaitMsg** allows **SendMessage** to return to the caller by using a Win32 PeekMessage loop to do message processing.

‹Previous Home Topic Contents Index Next›

CAMSchedule Class

CAMSchedule

The **CAMSchedule** class relieves clocks from the burden of managing the advise requests. A clock can delegate such management to this class, provided that it calls this class's Advise method when the earliest event should be fired. The application can fetch the time of the earliest event by calling GetNextAdviseTime, or the application can track events by a combination of the times returned by **Advise** and the event times that the clock adds.

Member Functions

Name	Description
AddAdvisePacket	Creates a new pending notification.
Advise	Requests the scheduler to dispatch all events up to and including the time specified.
CAMSchedule	Constructs a CAMSchedule object.
GetAdviseCount	Returns the number of outstanding events.
GetEvent	Returns the event handle to send if the advise time requires reevaluation.
GetNextAdviseTime	Returns the reference time at which the next advise should be set, or MAX_TIME if no events are scheduled.
Unadvise	Removes a previously established advise link.

‹Previous Home Topic Contents Index Next›

‹Previous Home Topic Contents Index Next›

CAMSchedule::AddAdvisePacket

CAMSchedule Class

Creates a new pending notification and adds it to the advise notification list.

DWORD AddAdvisePacket(
  const REFERENCE_TIME & *time1*,
  const REFERENCE_TIME & *time2*,
  HANDLE *hNotify*,
  BOOL *bPeriodic*

);

Parameters

time1
Time that the advise should take place.
time2
Period between notifications. (Ignored if *bPeriodic* is FALSE.)
hNotify
Notification mechanism. Either a semaphore handle (if *bPeriodic* is TRUE) or an event handle.
bPeriodic
Flag that specifies whether the notification is sent repeatedly, or whether the notification is sent once. This can be one of the following values:

Value	Meaning
TRUE	This is a periodic timer that will fire every *time2* units until canceled.
FALSE	This is a one-shot timer.

Return Values

Returns the advise token if successful, or zero if an error occurred.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

CAMSchedule::Advise

CAMSchedule Class

Requests the scheduler to dispatch all events up to and including the time specified. This method is expected to be called by a controlling clock specifying the current time, just in time to dispatch the next advise request.

REFERENCE_TIME Advise(
const REFERENCE_TIME & *rtTime*
);

Parameters

rtTime
Current reference time.

Return Values

Returns the reference time at which the next advise will expire, or MAX_TIME if there are no outstanding events.

Remarks

Clocks can call this method to advise the scheduler of the time. The scheduler will then signal all the events that have expired, and reschedule the periodic ones.

It is not intended that clocks should call this method all the time, rather that clocks will call **Advise** just one time. The time returned will be invalidated if you start adding extra advises.

Previous Home Topic Contents Index Next

CAMSchedule::CAMSchedule

CAMSchedule Class

Constructs a CAMSchedule object.

CAMSchedule(
HANDLE *hEvent*
);

Parameters

hEvent
Event that CAMSchedule should fire if the advise time needs reevaluating.

Return Values

No return value.

Previous Home Topic Contents Index Next

CAMSchedule::GetAdviseCount

CAMSchedule Class

Returns the number of outstanding events.

DWORD GetAdviseCount( );

Return Values

Returns the number of outstanding events.

Previous Home Topic Contents Index Next

CAMSchedule::GetEvent

CAMSchedule Class

Retrieves the event handle to set if the advise time requires reevaluation.

HANDLE GetEvent( );

Return Values

Returns a HANDLE to the event to set when this object's advise time requires reevaluation.

Previous Home Topic Contents Index Next

CAMSchedule::GetNextAdviseTime

CAMSchedule Class

Checks the time of the next advise.

REFERENCE_TIME GetNextAdviseTime( );

Return Values

Returns the reference time at which the next advise should be set, or MAX_TIME if there are no events scheduled.

CAMSchedule::Unadvise

CAMSchedule Class

Removes a previously established advise link.

HRESULT Unadvise(
DWORD *dwAdviseCookie*
);

Parameters

dwAdviseToken
Identifier (cookie) of the link that is being reset. This is the value returned by CAMSchedule::AddAdvisePacket.

Return Values

Returns S_OK if successful; otherwise, returns S_FALSE.

Remarks

This member function is modeled after the IReferenceClock::Unadvise method. Call **Unadvise** to remove the previously established clock advise links.

Unadvise should be called for unexpired single-shot advise requests. Calling **Unadvise** with the token of an already expired event causes no problems, so applications can choose to always call **Unadvise** on their single-shot events without fear of problems.

CAMThread Class

CAMThread

CAMThread is an abstract class, a worker thread class that provides creation, synchronization, and communication with a worker thread. The worker thread can be accessed from several client threads. The class provides member functions to create the thread, pass commands to it, and wait for it to exit.

Use a CCritSec object to ensure that only one thread can make a request at a time. Use two CAMEvent objects: one to signal to the worker that a request is outstanding, and the other to signal to the client thread that the request has been completed. A nonblocking CAMThread::CheckRequest member function allows the worker thread to check for new requests while working asynchronously.

Derive from this class to provide your own thread member function. You might also want to provide type-safe signaling member functions that package parameters and return values using the CAMThread::CallWorker member function.

Thread creation is independent of object creation. Create a member variable derived from **CAMThread**, and then use the member functions to start and stop the thread when needed.

Data Members

Name	Description
m_AccessLock	Critical section object that locks access by client threads.
m_WorkerLock	Critical section object that locks access to shared objects.

Member Functions

Name	Description
CallWorker	Makes a request to the worker thread.
CAMThread	Constructs a CAMThread object.
CheckRequest	Determines if there is an outstanding request. This is a nonblocking member function.
Close	Blocks until the thread has exited and released its resources.
Create	Starts the thread running.
GetRequest	Blocks until the next request is made and then returns a DWORD value.
GetRequestHandle	Returns an event handle.
GetRequestParam	Returns the latest request.
InitialThreadProc	Retrieves a **this** pointer. Carry out this member function before calling the CAMThread::ThreadProc member function.
Reply	Returns a DWORD value to the requesting thread and releases it, signaling completion of the request.
ThreadExists	Determines whether a thread exists or has exited.

ThreadProc	Indicates a pure virtual member function that is called on the worker thread.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

CAMThread::CallWorker

CAMThread Class

Makes a request to the worker thread and blocks for a response.

```
DWORD CallWorker(
  DWORD dw
  );
```

Parameters

dw
Derived class defines the meaning of the parameter.

Return Values

Returns a value that is defined by the derived class.

Remarks

This member function uses a CCritSec object to ensure that only one request is made at a time. It is therefore not valid to call the **CAMThread::CallWorker** member function from the thread itself or from any member function that is executing in the context of the thread.

Previous Home Topic Contents Index Next

CAMThread::CAMThread

CAMThread Class

Constructs a CAMThread object.

CAMThread( );

Return Values

No return value.

Remarks

Creates a CAMThread object but does not create an actual thread. You call the CAMThread::Create member function to create a thread.

Previous Home Topic Contents Index Next

CAMThread::CheckRequest

CAMThread Class

Determines if there is an outstanding request. This is a nonblocking member function.

BOOL CheckRequest(
DWORD *pParam*
);

Parameters

pParam
Parameter that assumes the value passed by the last call to the CAMThread::CallWorker member function.

Return Values

Returns TRUE if an outstanding request is still active, or FALSE is no request is active.

Remarks

If there is an outstanding request, the requesting thread will block until the CAMThread::GetRequest member function is called. The request remains outstanding (that is, this member function continues to return TRUE) until either the CAMThread::Reply or **CAMThread::GetRequest** member function is called.

Previous Home Topic Contents Index Next

CAMThread::Close

CAMThread Class

Blocks until the thread has exited and released its resources.

void Close(void);

Return Values

No return value.

Remarks

You must instruct the thread to exit by some other means; for example, call the CAMThread::CallWorker member function with a request that is interpreted by the derived class to mean complete and exit.

If the thread is still running when the CAMThread object is destroyed, the **CAMThread::Close** member function is called internally.

Previous Home Topic Contents Index Next

CAMThread::Create

CAMThread Class

Starts the thread running.

BOOL Create(void);

Return Values

Returns TRUE if the thread started successfully, or FALSE if the thread is already running.

Remarks

This member function creates the thread and calls the CAMThread::ThreadProc member function from the derived class.

Previous Home Topic Contents Index Next

CAMThread::GetRequest

CAMThread Class

Blocks until the next request is made.

DWORD GetRequest( );

Return Values

Returns a value that is defined by the derived class.

Remarks

This member function blocks the requesting thread until the CAMThread::Reply function is called.

Previous Home Topic Contents Index Next

CAMThread::GetRequestHandle

CAMThread Class

Returns an event handle for performance improvements.

HANDLE GetRequestHandle( ) const;

Return Values

Returns an event handle.

Remarks

To use the Microsoft Win32 WaitForMultipleObjects function, you will need this handle in the thread's wait list or the thread will not be responsive.

Previous Home Topic Contents Index Next

CAMThread::GetRequestParam

CAMThread Class

Returns the most recent request.

DWORD GetRequestParam( ) const;

Return Values

Returns a DWORD value that indicates the request made previously by the CAMThread::GetRequest member function.

Previous Home Topic Contents Index Next

CAMThread::InitialThreadProc

CAMThread Class

Receives a **this** pointer and calls the CAMThread::ThreadProc member function.

DWORD InitialThreadProc(
LPVOID *pv*
);

Parameters

pv
The **this** pointer.

Return Values

Returns the DWORD returned by CAMThread::ThreadProc. This **DWORD** is not defined by this class.

Previous Home Topic Contents Index Next

CAMThread::Reply

CAMThread Class

Returns a DWORD value to the requesting thread and releases it, signaling completion of the request.

void Reply(
DWORD *dw*
);

Parameters

dw
Value returned by the CAMThread::CallWorker member function on the client side.

Return Values

No return value.

Previous Home Topic Contents Index Next

CAMThread::ThreadExists

CAMThread Class

Determines whether the thread has been created and has not yet exited.

BOOL ThreadExists( );

Return Values

Returns TRUE if the thread exists and hasn't exited, or FALSE if the thread doesn't exist.

Previous Home Topic Contents Index Next