Logitech QuickCam® SDK 1.0
Microsoft® Win32 C++ Programmer's Guide

Logitech Confidential

QuickCam Team
Logitech, USA

Contents

P A R T I

QuickCam® SDK

chapter one

Introduction

With the emergence of new technology such as MPEG4 and Internet video broadcasting, computer cameras have finally become more of a tool than a novelty. Logitech, the world leader in computer camera sales, decided to provide a useful tool for software developers to easily control and utilize its family of QuickCam® cameras. From this decision, the QuickCam® SDK was created. In the following pages you will learn how to develop your own camera-based application, without the need to learn complex low-level camera interfaces. This document includes information about developing camera-based applications in C++.

1.1 Purpose

The Logitech QuickCam® SDK is your backstage pass to developing digital camera applications for Logitech QuickCam cameras. Traditional camera-based applications include complicated Video for Windows (VFW) or DirectShow® programming instructions. Additionally, from these interfaces only one camera connection is allowed at any given time. The Logitech QuickCam® SDK solves this single-connection issue and allows any number of simultaneous camera connections through an easy-to-use COM interface.

1.2 Features

- **Window-based User Interface**
  A windowed user-interface with live video preview and easy to access camera settings is available. Additionally, each user-interface element can be hidden or shown programmatically.

- **Access to Camera Settings**
  Modify brightness, contrast, hue, exposure, image size, and many other camera settings programmatically, or with a provided camera settings dialog.

- **Access to Real-Time video data**
  For power users, access to real-time video data is available. This powerful feature allows fast, direct access to video data. You can develop your own motion detector, or stream video data through a video conferencing engine easily with this feature.

- **Movie Recording**
  Recording movies is easy with numerous recording properties at your disposal. Change the frame-rate, video compressor, audio compressor, and other properties with simple COM properties.

- **Taking Pictures**
  Pictures can be saved either to disk or to an application supplied memory buffer.

- **Text Overlay**
  Program-specified text can be overlaid on pictures and movies. You can select text color, opaque or transparent backgrounds, font, font size, and many other options.

- **Camera Notifications**
  Receive camera plug and un-plug notifications as well as a variety of other useful notification events. These events help insure that your application is up-to-date with camera activity at all times.

- **Multiple Simultaneous Camera Connections**
  This power feature is the primary reason the QuickCam SDK exists. The QuickCam SDK creates virtual camera instances for a single camera both efficiently and transparently. This feature allows you, as the developer, to create multiple simultaneous connections to a camera without worry.

- **Motion Detection**
  Easily incorporate motion detection into your application. This bonus feature, once enabled, notifies your application with motion analysis information.

1.3 Where Applicable

The Logitech QuickCam® SDK allows programmers working in C++, MFC, Microsoft Visual Basic® and other high-level languages to easily develop high-performance camera-based applications. There are basically three approaches to programming a digital video camera under Microsoft Windows®:

1. Video for Windows®

    + De facto standard for video capture on Windows®.
    - No access to camera-specific features.
    - Not easy to learn or use.
    - Minimal: Many common tasks require additional programming.

2. DirectShow®

    - Not easy to learn or use.
    - Minimal: Many common tasks require additional programming.

3. Logitech QuickCam SDK

    + Full support of all camera features.
    + Highly accessible from many languages.
    + Simple interface to learn and use.
    + Advanced features such as text overlaying and motion detection.
    + Allows multiple simultaneous camera connections

    – Proprietary: Will only work with Logitech QuickCam® products.

1.4 Run-Time Requirements

The Logitech QuickCam SDK can be used in the following environments:

- Windows® 98
- Windows® 2000
- Windows® Millenium

Minimum System Requirements:

- Logitech QuickCam® video camera
- Pentium® 200 Mhz MMX (266 MHz or higher and MMX technology recommended)
- 16 MB RAM
- Hard Drive with 100 MB free space
- 16-bit color displays (thousands of colors)
- Windows-compatible sound card (full duplex sound card recommended) and mouse
- Speakers required for receiving audio

1.5 QuickCam® SDK Software Architecture

The QuickCam® SDK depends on the interaction of several components at run-time, although your application only communicates with one of them.

- **Logitech QuickCam® Video Portal Component**
  This component implements the COM interface described in this document. It is a windowed in-process COM object with a video preview area and various UI elements any of which can be enabled or disabled at run-time. Complete camera control is available including contrast, brightness, and gain. Videos can be recorded using any Video for Windows installable compressor. Pictures can be saved to either memory or disk. Direct real-time access to video streaming data is also available. There are many other features and subsets of the above features available from the video portal.

- **Logitech QuickCam® Video Server Component**
  The Video Server is invisible to your application. It serves to coordinate between multiple instances of the Video Portal, and to manage and centralize their communication with the Video for Windows subsystem. The Video Server also communicates with another Logitech component called LVCom, which communicates directly with the WDM camera driver.

The Video Server interface is not described in this document. Future versions of the QuickCam® SDK may contain information on the Video Server interface.

LVCom is installed as part of the camera product software, you do not have to include it or install it with your application.

The following figure shows the various camera software components and how they are related at run-time:

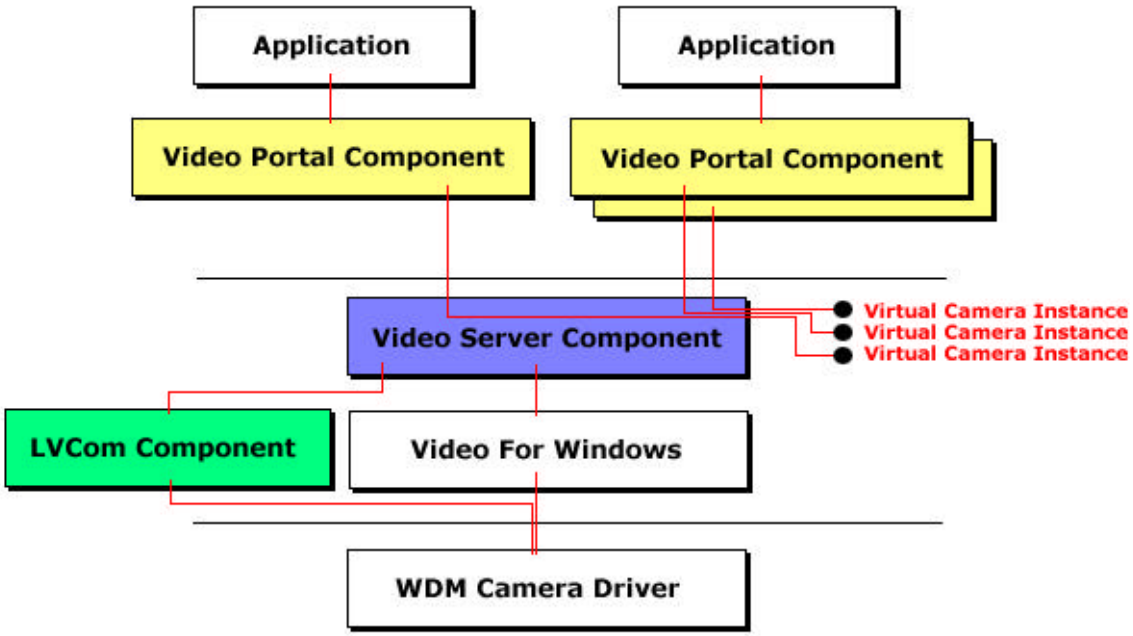


Figure 1 – Camera Software Components At Run-Time

chapter two

Getting Started with QuickCam® SDK

2.1 Installing the SDK

If you are reading this, there is a good chance you have already installed the QuickCam® SDK. If you haven't, you will need to obtain the SDK from Logitech. It is packaged as a downloadable self-extracting archive file: QCSDK1.EXE.

The QuickCam® SDK can be downloaded from the Logitech website http:://developer.logitech.com. Follow the instructions from this web site to install the SDK. If you have any questions or problems installing the QuickCam® SDK, see the http://developer.logitech.com web site for assistance.

Once the QuickCam® SDK is installed the following folders are created:

```
\QCSDK1
        \Docs
        \Inc
        \Redist
        \Samples
                \MFC
                    \BadgeMaker
                    \HydraQC
                    \StudioZero
                    \TextOver
                    \Vidbert
                    \bin
                \Visual Basic
                    \VBDemo
                    \VBMotion
                    \bin
                \Win32
                    \Feedback
                    \bin
```

2.2 Removing the SDK

To remove the QuickCam® SDK, use the **Control Panel** "Add/Remove Programs" feature of Windows.

2.3 Sample Code

The SDK includes sample code for Microsoft Visual C++® (Win32), Microsoft Visual C++® (MFC), and Microsoft Visual Basic® 5.0. All of this code is in \QCSDK1\Samples, in subdirectories named **WIN32**, **MFC** and **Visual Basic** respectively. Compiling the C++ (Win32) and MFC samples requires Microsoft Visual C++® 6.0. Compiling the Visual Basic samples require Microsoft Visual Basic® 5.0.

Microsoft Visual C++® (Win32) sample application

There is one **C++ (Win32)** sample that demonstrates using the QuickCam® SDK without a MFC (Microsoft Foundation Classes) dependency: Feedback. This sample includes header files from \QCSDK1\inc.

- **Feedback** – This C++ (Win32) sample demonstrates the QuickCam® SDK's motion detection capabilities. A simple motion magnitude graph is drawn depending on the amount of motion detected.

Microsoft Visual C++® (MFC) sample applications

There are five **MFC** samples that demonstrate using the QuickCam SDK via MFC (Microsoft Foundation Classes).

- **BadgeMaker** – This MFC sample demonstrates using the QuickCam SDK's to create photo ID badges. This sample uses the **PictureToMemory** method to save a picture into a memory buffer and overlay a text-stamp date onto the picture.
- **HydraQC** – This MFC sample demonstrates using the QuickCam SDK's to dynamically switch between multiple cameras.
- **StudioZero**– This MFC sample demonstrates using the QuickCam SDK's to record a movie.
- **TextOver**– This MFC sample demonstrates the QuickCam SDK's text overlay properties to apply a textual string to a picture.
- **Vidbert**– This MFC sample demonstrates the QuickCam SDK's video notification callback mechanism (see **StartVideoHook**). Various effects modify the live video image received in the video callback.

Microsoft Visual Basic® sample applications

There are two **Visual Basic** samples that demonstrate using the QuickCam SDK.

- **VBDemo** – This Visual Basic sample demonstrates the QuickCam SDK's motion detection capabilities. A simple motion magnitude graph is drawn depending on the amount of motion detected.
- **VBMotion** – This Visual Basic sample demonstrates several of the QuickCam SDK's features including movie recording, picture taking, and how to enable/disable the LED light on the camera.

2.4 Redistributable Files

The \QCSDK1\Redist folder of the QuickCam SDK contains the installation program and necessary files to re-distribute the QuickCam SDK. This setup program must be executed on all machines which use the QuickCam® SDK.

P A R T I I

QuickCam® Video Portal API

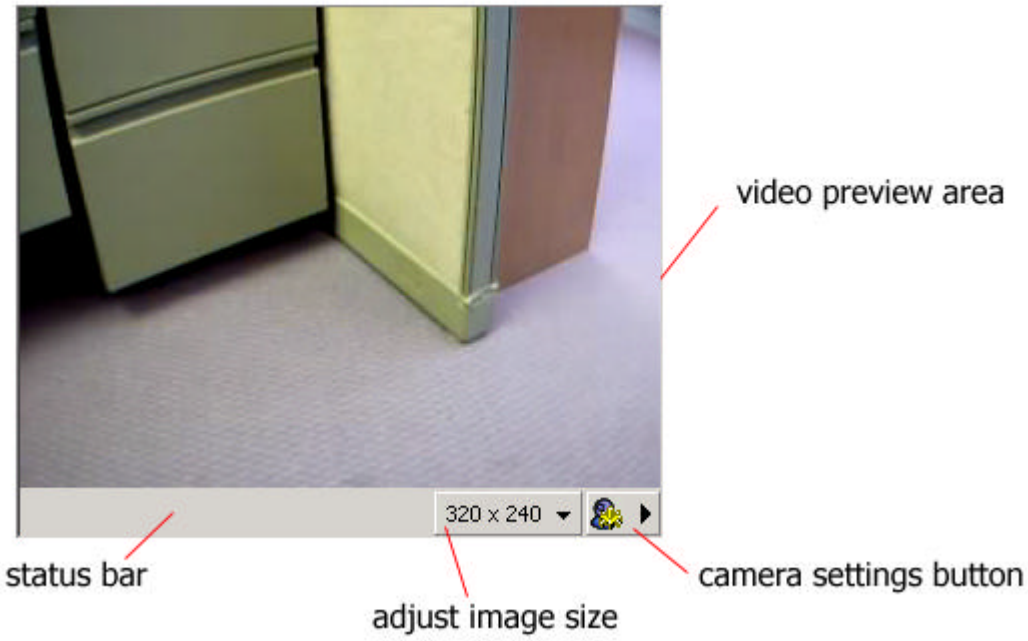
chapter three

Using the Video Portal API

This chapter begins with a tour of the visual aspects of the video portal control and continues on with some basic functionality. In this chapter you will learn how to create an instance of the video portal and start previewing live video. Examples of enabling and disabling user-interface elements are addressed. We also discuss saving images to disk and memory, recording movies, and creating text overlays.

3.1 User Interface Tour

The following diagram shows an example video portal window. There are only two main visual elements for the portal window: the preview area, and the status bar. The preview area is where live video is displayed. Within the status bar there are three main elements: a text area, an image size button, and a camera settings button. The text area of the status bar can show various actions in progress. Additionally, an application can send specific text messages to be displayed in this area. The image size button allows quick and easy access to modifying the image size received from the camera. Finally, the camera settings button allows users to quick access to camera settings from camera selection to adjusting contrast.



3.2 Getting Started

This is where we begin examining how to instantiate the video portal control and display live video on the screen. Getting video started with the video portal is easy. However, instantiating an ATL COM object without MFC is not quite so simple. However, the source code to instantiate a windowed ATL COM object is included. If you can look past the ATL glue, you'll realize that there are only a few lines of code to get video up and running in your application.

1. Our first step however is to create a **WIN32 Application** project using Microsoft Visual Studio® 6.0. Create the project based upon a "*typical Hello World! application*".

2. Next, insert the following lines into **stdafx.h**. These files are needed to use ATL com objects.

```
#include <atlbase.h>
extern CComModule _Module;
#include <atlcom.h>
#include <atlhost.h>

#include <ocidl.h>
#include <assert.h>
```

3. The video control depends on including the “LVServerDefs.H” header file. LVServerDefs.H is located in \QCSDK1\inc. Specify the \QCSDK1\inc directory in the preprocessor “Additional include directories” setting. Select the Project->Settings menu item to display the “Project Settings” dialog box. Select the C/C++ tab and choose “Preprocessor” in the category combo box. In the “Additional include directories” edit box, add the following: **\QCSDK1\inc**

4. Insert the following lines in bolded face into your projects implementation .CPP file.

```
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK    About(HWND, UINT, WPARAM, LPARAM);

. . .
// the following two files are located in the inc directory of the QuickCam
// SDK.  These two files contain interface definitions for the video portal
// control.
#include "VPortal2.h"
#include "VPortal2_i.c"
#include “LVServerDefs.H”
// IVideoPortal is the interface to the video portal control
IVideoPortal* gpVideo = NULL;

// InitializeVideo is used to actually create an instance of the video
// portal control and obtain a IVideoPortal pointer to this interface.
BOOL InitializeVideo( HWND hWnd );

// UnInitializeVideo is used for clean up purposes
BOOL UnInitializeVideo( void );

CComModule  _Module;
HWND        gMainHwnd = NULL;
DWORD       gdwAdviseCookie = 0;

//the following CDriver class implements the connection point to the video
//portal control.  This connection point allows the video control to commuicate
//with the application through the PortalNotification event.  The
//PortalNotification method is called by the video portal control and is
//handled here.

class CDriver :
    public IDispatchImpl<_IVideoPortalEvents, &IID__IVideoPortalEvents,
&LIBID_VPORTAL2Lib>,
    public CComObjectRoot
{
public:
    CDriver() {}
BEGIN_COM_MAP(CDriver)
    COM_INTERFACE_ENTRY(IDispatch)
    COM_INTERFACE_ENTRY(_IVideoPortalEvents)
END_COM_MAP()

    STDMETHOD(PortalNotification)(
            long lMsg,
            long lParam1,
```

```
            long lParam2,
            long lParam3)
    {
        // implement any PortalNotification notification handling here.
        switch( lMsg )
        {
        case NOTIFICATIONMSG_MOTION:
            break;
        case NOTIFICATIONMSG_MOVIERECORDERROR:
            break;
        case NOTIFICATIONMSG_CAMERADETACHED:
            break;
        case NOTIFICATIONMSG_CAMERAREATTACHED:
            break;
        case NOTIFICATIONMSG_IMAGESIZECHANGE:
            break;
        case NOTIFICATIONMSG_CAMERAPRECHANGE:
            break;
        case NOTIFICATIONMSG_CAMERACHANGEFAILED:
            break;
        case NOTIFICATIONMSG_POSTCAMERACHANGED:
            break;
        case NOTIFICATIONMSG_CAMERBUTTONCLICKED:
            break;
        case NOTIFICATIONMSG_VIDEOHOOK:
            break;
        case NOTIFICATIONMSG_SETTINGDLGCLOSED:
            break;
        case NOTIFICATIONMSG_QUERYPRECAMERAMODIFICATION:
            break;
        case NOTIFICATIONMSG_MOVIESIZE:
            break;
        default:
            break;
            };
        return S_OK;
    }

};

// the following defines the connection point interface pointer
CComObject<CDriver>* gpDriver;


int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{
. . .
```

5. Insert the following lines in bolded face into the **WinMain** function.

```
int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{
    //Initializes the COM library on the current apartment and identifies the
    //concurrency model as single-thread apartment (STA).
    CoInitialize(NULL);

    // Initialize the ATL module
```

```
    _Module.Init(NULL, hInstance);

    //Initialize ATL control containment code.
    AtlAxWinInit();


    // TODO: Place code here.
    MSG msg;
    HACCEL hAccelTable;

    // Initialize global strings
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    LoadString(hInstance, IDC_CONTROLTEST_X, szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance);

    // Perform application initialization:
    if (!InitInstance (hInstance, nCmdShow))
    {
        return FALSE;
    }

    hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_CONTROLTEST_X);

    // Main message loop:
    while (GetMessage(&msg, NULL, 0, 0))
    {
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }

    UnInitializeVideo(); // clean up the video control...
    _Module.Term();
    CoUninitialize();

    return msg.wParam;
}
```

6. Insert the following lines into the **InitInstance** function.

```
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
   HWND hWnd;

   hInst = hInstance; // Store instance handle in our global variable

   hWnd = CreateWindow( szWindowClass,
                        szTitle,
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT,
                        0,
                        CW_USEDEFAULT,
                        0,
                        NULL,
                        NULL,
                        hInstance,
                        NULL);

   if (!hWnd)
```

```
    {
       return FALSE;
    }

    gMainHwnd = hWnd;
    InitializeVideo( hWnd );

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    return TRUE;
}
```

7. Insert the following function into the project's implementation file:

```
// IntializeVideo creates the video portal control instance, obtains the
// connection point object, and connects the video portal to a camera device.
BOOL InitializeVideo( HWND hParent )
{
    long lResult;

    USES_CONVERSION;

    LPOLESTR strGUIDVideoPortalActiveXControl = NULL;
    StringFromCLSID(CLSID_VideoPortal, &strGUIDVideoPortalActiveXControl);

    // first we create a window which contains the video portal control.
    // notice the class name as "AtlAxWin", and the window name
    // is the CLSID of the video portal control itself…
    HWND hWndCtl = ::CreateWindow( "AtlAxWin",
                                   OLE2T(strGUIDVideoPortalActiveXControl),
                                   WS_CHILD | WS_VISIBLE | WS_GROUP,
                                   0,
                                   0,
                                   400,
                                   400,
                                   hParent,
                                   NULL,
                                   ::GetModuleHandle(NULL),
                                   NULL );


    CoTaskMemFree(strGUIDVideoPortalActiveXControl );

    if(!hWndCtl)
    {
        return FALSE;
    }

    // the following method retrieves the IVideoPortal interface from the HWND
    // of the window we just created.
    if ( FAILED( AtlAxGetControl(hWndCtl, (IUnknown **)&gpVideo)) )
    {
        return FALSE;
    }

    // the following instantiates the CDriver connection point object
    CComObject<CDriver>::CreateInstance(&gpDriver);

    // the following assigns the CDriver connection point to the video
    // portal's connection point.
    if ( FAILED ( AtlAdvise(gpVideo, gpDriver->GetUnknown(),
IID__IVideoPortalEvents, &gdwAdviseCookie) ) )
    {
```

```
        gpVideo->Release();
        gpVideo = NULL;
        return FALSE;
    }

    // now we have everything we need to start communicating with the
    // video portal.  The gpVideo object is the interface to the video
    // portal
    WCHAR wstrKey[] = L"HKEY_CURRENT_USER\\Software\\TestApp1";
    WCHAR wstrUnique[] = L"Aaron";
    BSTR bStrKey = ::SysAllocString(wstrKey);
    BSTR bStrUnique = ::SysAllocString(wstrUnique);

    // the following line initializes the video control to be used by
    // the application
    if ( FAILED(gpVideo->PrepareControl( bStrUnique, bStrKey, 0, &lResult)) )
    {
        ::SysFreeString(bStrKey);
        ::SysFreeString(bStrUnique);

        UnInitializeVideo();
        return FALSE;
    }

    ::SysFreeString(bStrUnique);

    gpVideo->put_PreviewMaxWidth( 320 );
    gpVideo->put_PreviewMaxHeight( 240 );

    // next we turn on the video portal's status bar
    if ( FAILED(gpVideo->EnableUIElements( UIELEMENT_STATUSBAR,
                                           0,
                                           TRUE,
                                           &lResult )) )
    {
    }

    // next, we connect to a camera device
    if ( FAILED(gpVideo->ConnectCamera2( &lResult )) )
    {
        ::SysFreeString(bStrKey);

        UnInitializeVideo();
        return FALSE;
    }

    ::SysFreeString(bStrKey);

    // finally, we tell the video portal, to enable video preview.
    gpVideo->put_EnablePreview(TRUE);
    return FALSE;
}
```

8. Insert the following function into the project's implementation file:

```
// UnInitialize video performs cleanup of the video portal when
//it is no longer needed.
BOOL UnInitializeVideo( void )
{
    if ( gpDriver )
    {
        AtlUnadvise(gpVideo,IID__IVideoPortalEvents, gdwAdviseCookie);
        gpDriver = NULL;
```

```
    }

    if ( gpVideo )
    {
        gpVideo->Release();
        gpVideo = NULL;
    }

    return TRUE;
}
```

9. Next, compile and execute your project.

Let's take the major steps that actually demonstrate using the video control one by one.

Initialize the Video Portal Interface

The first video portal method called is **PrepareControl**. This method establishes a connection from the video portal to the video server. The **PrepareControl** method must always be called before any other video portal method. It specifies a name, a registry key, and initialization flags. The most interesting parameter is the name. All instances of the video portal with the same name share settings. For this reason, it is recommended that the name is carefully chosen. The video portal uses the registry key to save specific settings, such as image size and PictureSmart™ information. The video portal does not save global camera settings such as brightness, contrast and brightness.

Configure the Portal User Interface

Next, we call the **EnableUIElements** method to enable the status bar. There are several visual elements, which can be enabled or disabled. For instance, by default the following images sizes are supported: 160x120, 320x240, and 640x480. *(Note the QuickCam® Express camera does not support 640x480)*. You can forbid a user from switching to 320x240 by using the **EnablueUIElements** method and passing UIELEMENT_320x240, with a status of FALSE. For other examples, see the reference section on **EnableUIElements**.

Connect to a camera

We are now ready to establish a connection to a QuickCam® camera. The recommended method to do this is **ConnectCamera2**. It establishes a camera connection with any camera that is currently in use by another portal instance. If there is not a current camera connection, the last camera used is chosen. There is an additional method **ConnectCamera** that allows connecting to a specific camera. However, this method should be used with caution, since all video portal instances *share the same camera*. Connecting to a specific camera will cause all video portals to start using that camera.

Open the video preview window

Once a camera connection has been successfully established, we are ready to turn on the video preview display. The **put_EnablePreview** method enables video preview.

3.3 Taking Pictures

Next we examine how to use the video portal for saving pictures either to a file or into a memory buffer. First we will examine taking a picture and saving it to a file. The following code fragment demonstrates how this is accomplished.

```
{
    . . .

    WCHAR wFile[] = L"c:\\image_test.bmp";
    BSTR bStrFile = ::SysAllocString(wFile);
```

```
    long lResult;

    if ( FAILED( gpVideo->PictureToFile( 0, 24, bStrFile, NULL, &lResult ) ))
    {
        // handle error condition
    }

    ::SysFreeString(bStrFile);

}
```

Currently, the only supported format for saving images to both memory and disk is 24 bit RGB, see the reference section on **PictureToFile** for more information. Next we examine saving a picture to a memory buffer. The video portal saves into memory a.BMP file as it would exist on disk. In the prototype below, we first call **PictureToMemory** to receive the number of bytes required to store the image. After allocating a buffer of this size, we call **PictureToMemory** a second time to perform the actual save operation.

```
{
    . . .
    long lResult;
    long lSize;
    if ( FAILED(gpVideo->PictureToMemory(0, 24, 0, &lSize, NULL, &lResult)) )
    {
        return FALSE;
    }

    BYTE * pBuffer = new BYTE[lSize];

    if ( FAILED(gpVideo->PictureToMemory( 0, 24, (long)pBuffer, &lSize, NULL,
&lResult )) )
    {
        delete []pBuffer; // delete the memory we allocated...
        return FALSE;
    }
    .
    .
    //pBuffer now contains a .BMP file in memory
    .
    .
    delete []pBuffer;
    return TRUE;
}
```

3.4 Overlay Text

Our next task is to demonstrate how to overlay text on an image. You can specify color, font, font size, indicate whether or not to use a text shadow, and finally whether the overlay is opaque or transparent.

```
{
    . . .

    WCHAR wFile[] = L“c:\\image_test_overlay.bmp”;
    BSTR bStrFile = ::SysAllocString(wFile);

    WCHAR wDate[] = L“Monday April 4, 2000”;
    BSTR bStrDate = ::SysAllocString(wDate);

    WCHAR wFont[] = L“Arial”;
    BSTR bStrFont = ::SysAllocString(wFont);

    gpVideo->put_StampTextColor( (OLE_COLOR)RGB( 255,255,255) );
    gpVideo->put_StampFontName( bStrFont );
```

```
    gpVideo->put_StampPointSize( 10 );
    gpVideo->put_StampTextShadow( FALSE );
    gpVideo->put_StampTransparentBackground( TRUE );

    long lResult;

    gpVideo->PictureToFile( 0, 24, bStrFile, bStrDate, &lResult );

    ::SysFreeString(bStrFile);
    ::SysFreeString(bStrDate);
    ::SysFreeString(bStrFont);

}
```

The function prototype above demonstrates how to specify text stamp properties. Once the text stamp properties have been set, you do not need to re-set them for each picture. The image below depicts the image saved to disk with these parameters.

3.5 Recording Movies

Recording movies is a simple task when using the Video Portal. You have the option to specify a variety of recording properties. It should be noted here that only one movie can be recorded at any given time, and while a movie is being recorded, no pictures can be saved to either memory or to disk.

By default all movies are saved to disk at the fasted possible frame rate, which varies with the type of camera used.

The following codes illustrates how to start and stop a movie recording session. This example is fairly straightforward and uses the default recording properties supplied by the Video Portal.

```
void StartMovie( void )
{
    long lResult;

    WCHAR wFile[] = L"c:\\sample_movie_1.avi";
    BSTR bStrFile = ::SysAllocString(wFile);


    if ( FAILED(gpVideo->StartMovieRecording(bStrFile, NULL, & lResult )))
```

```
    {
        ::SysFreeString(bStrFile);
        return;
    }

    ::SysFreeString(bStrFile);
}

void OnStopMovie( void )
{
    gpVideo->StopMovieRecording();
}
```

3.6 Animation (step-capture, time-lapse) Movie Recording

Our next example is slightly more difficult than the one above. We take a look at how to record a 'stop motion' or 'time-lapse' movie. Essentially, it's the same process as above, except a Video Portal method must be called for each frame to add to the movie. The Video Portal maintains a *recording-mode*. In the example above the mode was fast-as-possible, which is its default mode. For stop-motion and time-lapse recording, we use the *manual triggered* mode.

```
void OnStartStepCapture( void )
{
    long lResult;
    WCHAR wFile[] = L"c:\\sample_stepcapture.avi";
    BSTR bStrFile = ::SysAllocString(wFile);

    // Select the manual triggered recording mode:
    gpVideo->put_MovieRecordMode(STEPCAPTURE_MANUALTRIGGERED);

    // Specify the playback rate for this movie – 15 FPS:
    gpVideo->put_MoviePlaybackFPS( 15 );

    // Doesn't make sense to record audio in this mode:
    gpVideo->put_MovieRecordAudio(FALSE);

    if ( FAILED(gpVideo->StartMovieRecording(bStrFile, NULL, &lResult )))
    {
        ::SysFreeString(bStrFile);
        return;
    }
}

void OnAddFrame( void )
{
    gpVideo->StepCaptureAddFrame();
}

void OnStopMovie( void )
{
    gpVideo->StopMovieRecording();
}
```

chapter four

Video Portal Interface Reference

The pages that follow list each method, property and notification event for the video portal. This is the first SDK release of the video portal interface. Future versions of the video portal will be backward compatible with this interface.

Methods

PrepareControl
GetCameraCount
GetCameraDescription
GetCameraType
QueryCameraConnected
QueryRegistryCameraIndex
GetCameraState
ConnectCamera
ConnectCamera2
DisconnectCamera
LoadRegistrySettings
SaveRegistrySettings
SetCameraPropertyLong
GetCameraPropertyLong
PictureToFile
PictureToMemory
StartMovieRecording
StopMovieRecording
StepCaptureAddFrame
MovieRecordWriteSingleFrame
SetVideoFormat
GetVideoFormat
GetLastError
ShowCameraDlg
EnableUIElements
StartVideoHook
StopVideoHook

Properties

CameraConnected (get)
CameraIndex (get)
PictureSound (get/put)
StampTextColor (get/put)
StampFontName (get/put)
StampPointSize (get/put)
StampTextShadow (get/put)
StampTextShadowColor (get/put)
StampTransparentBackground (get/put)
StampBackgroundColor (get/put)
EnablePreview (get/put)
MovieVideoCompressionFOURCC (get/put)
MovieVideoCompressionKeyFrameInterval (get/put)
MovieVideoCompressionQuality (get/put)
MoviePlaybackFPS (get/put)
MovieAudioSamplesPerSecond (get/put)
MovieAudioChannels (get/put)
MovieAudioBitsPerSample (get/put)
MovieAudioCompressionFOURCC (get/put)
MovieAudioCompressionQuality (get/put)
MovieRecordAudio (get/put)
MovieRecordMode (get/put)
MovieCreateFlags (get/put)
MovieRecordingActiveLocal (get)
MovieRecordingActiveGlobal (get)
CameraState (get)
EnableMovieRecordErrorPrompt (get/put)
EnablePictureDiskErrorPrompt (get/put)
StatusBarText (put)
PreviewMaxWidth (get/put)
PreviewMaxHeight (get/put)

Notifications

PortalNotification

NOTIFICATIONMSG_MOTION
NOTIFICATIONMSG_MOVIERECORDERROR
NOTIFICATIONMSG_CAMERADETACHED
NOTIFICATIONMSG_CAMERAREATTACHED
NOTIFICATIONMSG_IMAGESIZECHANGE
NOTIFICATIONMSG_CAMERAPRECHANGE
NOTIFICATIONMSG_CAMERACHANGEFAILED
NOTIFICATIONMSG_POSTCAMERACHANGED
NOTIFICATIONMSG_CAMERBUTTONCLICKED
NOTIFICATIONMSG_VIDEOHOOK
NOTIFICATIONMSG_SETTINGDLGCLOSED
NOTIFICATIONMSG_QUERYPRECAMERAMODIFICATION
NOTIFICATIONMSG_MOVIESIZE

Methods

PrepareControl (Method)

The **PrepareControl** method initializes an instance of the Video Portal.

```
HRESULT PrepareControl(
        BSTR    strUniqueName,      // unique name for this portal instance
        BSTR    strRegistryKey,     // registry key location
        long    lFlags,             // initialization flags
        long*   plResult
        );
```

Parameters

strUniqueName

BSTR string that specifies a name for this *video portal* instance. Typically, each portal instance should be given a name which is unique to all portal instances. You can specify NULL to have the video portal generate a unique name for you.

strRegistryKey

BSTR string that specifies the registry location in which values for this portal will be saved. If you are not used to working with registry key values, you can specify "HKEY_CURRENT_USER\Software\Logitech**name**", where **name** specifies a string identifying your application. For example, "HKEY_CURRENT_USER\Software\Logitech\MyApp".

The following data is saved automatically in the registry under the specified key.

- PictureSmart™ settings
- Image size and depth
- Picture sound
- Text stamp settings
- Audio format settings
- Motion sensitivity setting

lFlags

This flag currently has no significance and must be 0.

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

This method must be called before any other video portal method. However, properties can be set and retrieved before calling this method. The video portal uses the registry-key parameter to save specific instance information, such as image size, etc.

Example

The following code fragment demonstrates using the PrepareControl method.

```
WCHAR wstrKey[] = L"HKEY_CURRENT_USER\\Software\\MyApp";
WCHAR wstrUnique[] = L"MyAppTest_001";

BSTR bStrKey = ::SysAllocString(wstrKey);
BSTR bStrUnique = ::SysAllocString(wstrUnique);

long lResult ;
if ( !FAILED(gpVideo->PrepareControl( bStrUnique, bStrKey, 0, &lResult ) )
{
    //unable to initialize video server component…
    ::SysFreeString(bStrKey);
    ::SysFreeString(bStrUnique);

}

::SysFreeString(bStrKey);
::SysFreeString(bStrUnique);
```

GetCameraCount (Method)

The **GetCameraCount** method retrieves the number of camera devices present.

```
HRESULT GetCameraCount(
      long* plCount,
      long* plResult
      );
```

Parameters

plCount

Points to a long that receives the number of current camera devices.

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

None.

Example

The following code fragment demonstrates using the GetCameraCount method.

```
long lCameraCount = 0;
long lResult;
if ( FAILED(gpVideo->GetCameraCount( &lCameraCount, &lResult) ))
    return ERROR;

printf("there are $ld cameras available", lCameraCount );
```

GetCameraDescription (Method)

The **GetCameraDescription** method retrieves the camera description at a specific device index.

```
HRESULT GetCameraDescription(
      long lIndex,
      BSTR* strDescription,
      long* plResult
      );
```

Parameters

lIndex

Specifies the index of the camera: A number from 0 to N-1, where N is the camera count returned by GetCameraCount.

strDescription

Points to a BSTR which receives the camera description.

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

This method is usually called when an application wishes to enumerate the cameras and connect to a specific device.

Example

The following code fragment demonstrates using the GetCameraDescription method.

```
long lCameraCount = 0;
long lResult;
if ( FAILED(gpVideo->GetCameraCount( &lCameraCount, &lResult) ))
    return ERROR;

char sBuffer[256];
for ( long i = 0 ; i < lCameraCount; i++ )
{
    BSTR bstrDesc = NULL;
    if ( FAILED(gpVideo->GetCameraDescription( i, &bstrDesc) ))
        continue;

    WideCharToMultiByte( CP_ACP, 0, bstrDesc, -1,
                         sBuffer, sizeof(sBuffer),
                         NULL,NULL);
    printf("camera name at index %ld is %s", i, sBuffer  );
}
```

GetCameraType (Method)

The **GetCameraType** method retrieves the type of a camera.

```
HRESULT GetCameraType(
      long lIndex,
      long* plCameraType,
      long* plResult
      );
```

Parameters

lIndex

Specifies the index of the camera: A number from 0 to N-1, where N is the camera count returned by GetCameraCount.

plCameraType

Receives the camera type at the specified camera index.

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

If this function is successful, the camera type will be one of the following values. These values are defined in LVServerDefs.H header file.

```
CAMERA_UNKNOWN = 0,
CAMERA_QUICKCAM_VC = 1,
CAMERA_QUICKCAM_QUICKCLIP =2,
CAMERA_QUICKCAM_PRO = 3,
CAMERA_QUICKCAM_HOME = 4,
CAMERA_QUICKCAM_PRO_B = 5,
CAMERA_QUICKCAM_TEKCOM = 6,
CAMERA_QUICKCAM_EXPRESS=7,
CAMERA_QUICKCAM_FROG=8,
CAMERA_QUICKCAM_EMERALD=9,
```

Example

The following code fragment demonstrates using the GetCameraType method to find the QuickCam Express camera index.

```
long lCameraCount = 0;
if ( FAILED(gpVideo->GetCameraCount( &lCameraCount, &lResult) ))
    return ERROR;

long lCameraType;
for ( long i = 0 ; i < lCameraCount; i++ )
{
    if ( FAILED(gpVideo->GetCameraType( i, &lCameraType, &lResult) ))
        continue;

    if ( (CAMERA_TYPE)lCameraType == CAMERA_QUICKCAM_EXPRESS )
        return SUCCESS;
}
```

QueryCameraConnected (Method)

The **QueryCameraConnected** determines if a camera at a specified index is connected already. The term “connected” refers to a driver instance connection, not a physical one. A camera “connection” exists when it is currently in use by an application.

```
HRESULT QueryCameraConnected(
      long lIndex,
      long* plConnected,
      long* plResult
      );
```

Parameters

lIndex

Specifies the index of the camera device.

plConnected

Receives a Boolean value of either TRUE (1) or FALSE (0).

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

This method can be used to determine if a camera is currently in use by another video portal before connecting this portal instance to it as well.

QueryRegistryCameraIndex (Method)

The **QueryRegistryCameraIndex** is used to retrieve the index of the camera from the registry.

```
HRESULT QueryRegistryCameraIndex(
      long* plIndex,
      long* plResult
      );
```

Parameters

plIndex

Receives the camera index from the registry.

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

This method can be used to determine the “last camera index used”, or likewise, the current one.

GetCameraState (Method)

The **GetCameraState** is used to get the state of the camera from the registry.

```
HRESULT GetCameraState(
      long lIndex,
      long* plCameraState,
      long* plResult
      );
```

Parameters

lIndex

A long which indicates the camera index to query.

plCameraState

Feceives the camera state.

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

The camera state can be any one of the following values.

CAMERA_OK = 0,
CAMERA_UNPLUGGED = 1,
CAMERA_INUSE = 2,
CAMERA_ERROR = 3,
CAMERA_SUSPENDED = 4,
CAMERA_UNKNOWNSTATUS = 10,

Example

The following code fragment demonstrates using the **GetCameraState** method to find the first camera which is unplugged (if any).

```
long lCameraCount = 0;
if ( FAILED(gpVideo->GetCameraCount( &lCameraCount, &lResult) ))
    return ERROR;

long lCameraState;
for ( long I = 0 ; I < lCameraCount; I++ )
{
    if ( FAILED(gpVideo->GetCameraState( I, & lCameraState) ))
        continue;

    if ( (CAMERA_STATUS)lCameraType == CAMERA_UNPLUGGED)
        return SUCCESS;
}
```

ConnectCamera (Method)

The **ConnectCamera** method establishes a connection to a specific camera device.

```
HRESULT ConnectCamera(
      long lIndex,
      long* plResult
      );
```

Parameters

lIndex

Specifies the index of the camera device to connect to.

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

Care should be used when calling this method. Connecting to a specific camera causes all other portal instances to start using the new camera. Consider using the **ConnectCamera2** method instead, it's designed to work in a friendlier manner.

Example

The following code fragment demonstrates using the **ConnectCamera** method to connect to a QuickCam Express.

```
long lCameraCount = 0;
long lResult;
if ( FAILED(gpVideo->GetCameraCount( &lCameraCount, lResult) ))
    return ERROR;

long lCameraType;
for ( long I = 0 ; I < lCameraCount; I++ )
{
    if ( FAILED(gpVideo->GetCameraType( I, &lCameraType, &lResult) ))
        continue;

    if ( (CAMERA_TYPE)lCameraType == CAMERA_QUICKCAM_EXPRESS )
    {
        if ( FAILED(gpVideo->Connect( I ) ))
            continue;

        return SUCCESS;
    }
}
return ERROR;       // no such camera found
```

ConnectCamera2 (Method)

The **ConnectCamera2** method establishes a connection to an appropriate camera device.

```
HRESULT ConnectCamera2(long* plResult);
```

Parameters

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

This method will connect to the last camera device used. If no camera connection has ever been established, the first valid camera will be connected. Otherwise, any valid camera will be connected.

This is the normal mechanism for connecting with a camera.

Example

The following code fragment demonstrates using the ConnectCamera2 method.

```
long lCameraCount = 0;
long lResult;

if ( FAILED(gpVideo->ConnectCamera2(&lResult) ))
    return ERROR;

return SUCCESS;
```

DisconnectCamera (Method)

The **DisconnectCamera** method removes the current camera connection.

```
HRESULT DisconnectCamera(long* plResult);
```

Parameters

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

Typically, this method is not called, when the control is released, the camera is automatically disconnected.

LoadRegistrySettings (Method)

The **LoadRegistrySettings** method loads instance-specific data from a registry key into the portal. Typically, this method is not used.

```
HRESULT LoadRegistrySettings(
      BSTR strRegistryKey,
      long* plResult
      );
```

Parameters

strRegistryKey

A BSTR string specifying a registry key.

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

Normally this method is not called, as the Video Portal automatically saves and loads registry settings as needed.

The following data is loaded from the registry using the specified key. If there is not data present or corrupt values are found, default settings are used.

- PictureSmart™ settings
- Image size and depth
- Picture sound
- Text stamp settings
- Audio format settings
- Motion sensitivity setting

SaveRegistrySettings (Method)

The **SaveRegistrySettings** method saves instance specific data into a registry key. Typically, this method is not used.

```
HRESULT SaveRegistrySettings(
      BSTR strRegistryKey,
      long* plResult

      );
```

Parameters

strRegistryKey

BSTR string which specifies a registry key.

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

Normally this method is not called, as the video control automatically saves and loads registry settings as it needs to.

The following data is saved into the registry using the specified key.

- PictureSmart™ settings
- Image size and depth
- Picture sound
- Text stamp settings
- Audio format settings
- Motion sensitivity setting

SetCameraPropertyLong (Method)

The **SetCameraPropertyLong** sets a camera-specific property.

```
HRESULT SetCameraPropertyLong(
      long lProperty,
      long lPropertyValue,
      long* plResult
      );
```

Parameters

lProperty

Specifies the property to set. See **Appendix A** for details.

lPropertyValue

Specifies the new value of the property.

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

This method is reserved for advanced users who wish to control features such as gain, brightness, and contrast manually. See **Appendix A** for information on special camera properties and the cameras supporting them.

Example

The following code fragment demonstrates using the **SetCameraProperty** method to turn on the LED light of the camera, if it is supported.

```
    long lPropertyValue;
    long lResult;

    if ( FAILED(gpVideo->GetCameraPropertyLong( PROPERTY_LED, & lPropertyValue,
&lResult) ))
        return PROPERTY_NOT_SUPPORTED;

    gpVideo->SetCameraPropertyLong( PROPERTY_LED, LED_ON, &lResult );
    return SUCCESS;
```

GetCameraPropertyLong (Method)

GetCameraPropertyLong gets the value of a camera-specific property.

```
HRESULT GetCameraPropertyLong(
      long lProperty,
      long* plPropertyValue,
      long* plResult
      );
```

Parameters

lProperty

Specifies the property to retrieve. See Appendix A for properties and property codes.

plPropertyValue

Points to a long that receives the property value.

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

Use this method to set specific camera property values. See **Appendix A** for information on camera properties and the cameras supporting them.

Example

The following code fragment demonstrates using the GetCameraProperty method to determine if PictureSmart™ is supported and enabled.

```
long lPropertyValue;
long lResult;
if ( FAILED(gpVideo->GetCameraPropertyLong(PROPERTY_PICTSMART_MODE,
                                    &lPropertyValue) ))
    return PROPERTY_NOT_SUPPORTED;

printf(" PictureSmart™ is %s enabled\n",
       lPropertyValue ? "currently", "not" );
```

PictureToFile (Method)

PictureToFile captures one frame from the currently connected camera writes it to a file.

```
HRESULT PictureToFile(
      long lFormatFourCC,
      long lBitDepth,
      BSTR strFileName,
      BSTR strTextStamp,
      long* plResult);
```

Parameters

lFormatFourCC

Specifies the format in which to save to picture. Currently, the only format supported is RGB24 uncompressed, so this parameter must be 0.

lBitDepth

Specifies the bit depth of the picture. Currently, the only bit-depth supported is 24.

strFileName

BSTR string indicating the file to save the picture to. You must specify the .bmp extension.

strTextStamp

BSTR string indicating a textual stamp to overlay on the picture. This parameter can be NULL.

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

Use this method to save an image to a file on disk. The only supported format is uncompressed 24-bit RGB, which is written to disk in Windows BMP format.

Example

```
SaveImageToFile(void)
{
    WCHAR wstrFile[] = L"image_test.bmp";
    BSTR bStrFile = ::SysAllocString(wstrFile);

    WCHAR wstrText[] = L"Hello.bmp";
    BSTR bStrText = ::SysAllocString(wstrText);

    gpVideo->PictureToFile( 0, 24, bStrFile, bStrText, &lResult );

    ::SysFreeString(bStrFile);
    ::SysFreeString(bStrText);

}
```

PictureToMemory (Method)

PictureToMemory gets an image from the camera and puts it into a buffer.

```
HRESULT PictureToMemory(
      long lFormatFourCC,
      long lBitDepth,
      long lMemory,
      long* plMemorySize,
      BSTR strTextStamp,
      long* plResult);
```

Parameters

lFormatFourCC

Specifies the format in which to save the image. Currently, the only format supported is RGB24 uncompressed, so this parameter must be 0.

lBitDepth

Specifies the bit depth of the image. Currently, the only bit depth supported is 24.

lMemory

Points to a buffer in memory where the image will be written (can be NULL, read on.)

plMemorySize

Points to a long that receives the size of the image data. If the lMemory parameter is 0 (NULL), this parameter can be used to determine how many bytes to allocate.

strTextStamp

BSTR string indicating a textual stamp to overlay on the picture. This parameter can be NULL.

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

If lMemory is not NULL, an image is captured and stored *as a BMP file* in memory starting at lMemory. Regardless of the value of lMemory, the long pointed to by plMemorySize is set to the number of bytes needed to hold the captured image data. Additionally, plMemorySize must point to the size of the buffer pointed to by lMemory. If the size pointed to by plMemorySize is not large enough to store the image, this method will fail.

Example

```
BYTE* pBuffer;
long lSize;
long lResult;

WCHAR wstrText[] = L"Hello Mom";
BSTR bStrText = ::SysAllocString(wstrText);
```

```
    gpVideo->PictureToMemory(0, 24, NULL, &lSize, bStrText, &lResult );

    pBuffer = malloc(lSize); // allocate buffer
    if (pBuffer) {
        // Got a buffer, capture frame into it
        gpVideo->PictureToMemory(0, 24, (long)pBuffer, &lSize, bStrText,
&lResult );
        . . .
        free(pBuffer);
    }

    ::SysFreeString(bStrText);
```

StartMovieRecording (Method)

The **StartMovieRecording** inititiates movie capture.

```
HRESULT StartMovieRecording(
      BSTR strFileName,
      BSTR strTextStamp,
      long* plResult
);
```

Parameters

strFileName

BSTR string indicating the location in which to save the movie to. For example: “c:\\windows\\desktop\\aaron.avi”. The filename specified must contain the extension “.avi".

strTextStamp

Not currently implemented, must be NULL.

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

Typically, an application sets the movie recording mode, FPS, compressor, compressor quality, audio settings, etc., before calling this method. Recording continues until StopMovieRecording is called or an error occurs. When can error during a movie recording session which causes the recording to stop, the **NOTIFICATIONMSG_MOVIERECORDERROR** notification message is sent to the container application. For more information on receiving notification messages, see the section on **Notifications**.

StopMovieRecording (Method)

The **StopMovieRecording** method ends movie capture.

```
HRESULT StopMovieRecording(long *plResult);
```

Parameters

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

None.

StepCaptureAddFrame (Method)

The **StepCaptureAddFrame** method adds a frame to a manual step capture movie.

```
HRESULT StepCaptureAddFrame(long* plResult);
```

Parameters

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

This method is only available during a movie recording session when the movie mode is STEPCAPTURE_MANUALTRIGGERED.

MovieRecordWriteSingleFrame (Method)

MovieRecordWriteSingleFrame opens an AVI file, appends a single frame, and closes the file.

```
HRESULT MovieRecordWriteSingleFrame(
      BSTR strFileName,
      BSTR strTextStamp,
      long* plResult
      );
```

Parameters

strFileName

BSTR string indicating the file to append to. For example: "c:\\windows\\desktop\\sample.avi"

strTextStamp

Not currently implemented, must be NULL.

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

This method is suitable for creating a time-lapse movie, where there are at least a few seconds between frames. Because of the overhead of opening and closing the AVI file, this method will always be much slower than **StepCaptureAddFrame**.

GetLastError (Method)

GetLastError retrieves the last Video Portal error. **Currently, this method is not implemented.**

```
HRESULT GetLastError(
      long* plError
      );
```

Parameters

plError

Points to a long that receives the last error generated or detected within this Video Portal instance.

Return Values

S_OK = Success; All other values = Failure.

Remarks

Use this method to retrieve the last generated error. **Currently, this method will always return 0.**

SetVideoFormat (Method)

SetVideoFormat sets the specific format that the video control preview will use.

```
HRESULT SetVideoFormat(
      long lWidth,
      long lHeight,
      long lBitDepth,
      long lFormatFourCC
      long* plResult
      );
```

Parameters

lWidth

Specifies the **width** in pixels of the video. (Note 1)

lHeight

Specifies the **height** in pixels of the video. (Note 1)

lBitDepth

Specifies the **bit depth** of the video. Currently, the only supported bit depth is 24.

lFormatFourCC

Specifies the video **format**. Currently, the only supported format is RGB24, indicated by code 0.

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

Use this method to specify the format of the video used by the control. The default format for the video portal is 320x240, 24Bit RGB. However, once the video portal has been created, it will maintain the last image size used the next time the portal is created.

Note 1: The only currently allowed sizes are: 160x120, 320x240, 640x480.
Note 2: All movies and pictures are saved using the current image size.

GetVideoFormat (Method)

GetVideoFormat retrieves the current video format used by the video control.

```
HRESULT GetVideoFormat(
      long* plWidth,
      long* plHeight,
      long* plBitDepth,
      long* lFormatFourCC,
      long* plResult
      );
```

Parameters

plWidth

Points to a long value that receives the **width** in pixels of the video.

plHeight

Points to a long value that receives the **height** in pixels of the video.

plBitDepth

Points to a long value that receives the **bit depth** of the video.

plFormatFourCC

Points to a long value that receives the four-character code specifying the **format** of the video.

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

The only possible video sizes are: 160x120, 320x240, 640x480
The only possible bit depth is: 24.
The only possible format is: uncompressed RGB (code 0).

ShowCameraDlg (Method)

ShowCameraDlg invokes the camera settings dialog.

```
HRESULT ShowCameraDlg(long* plResult);
```

Parameters

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

Typically this method is not used. Right clicking on the portal window opens the camera settings dialog box. Additionally, when the status-bar is enabled, there is an icon which launches the camera settings dialog box.

EnableUIElements (Method)

EnableUIElements enables and disables elements of the video portal user interface.

```
HRESULT EnableUIElements(
      long lElement,
      long lFlags,
      long lEnable,
      long* plResult
      );
```

Parameters

lElements

Specifies a UI element As follows:

`UIELEMENT_640x480 = 0`
This UI element indicates whether or not 640x480 is valid in the portal instance.

`UIELEMENT_320x240 = 1`
This UI element indicates whether or not 320x240 is valid in the portal instance.

`UIELEMENT_PCSMART = 2`
This UI element indicates whether or not the PictureSmart™ is valid in the portal instance.

`UIELEMENT_STATUSBAR = 3`
This UI element indicates whether or not the status bar should be displayed below the portal window. Note that if the status bar is not enabled, there is no point to enabling UIELEMENT_UI below, because the camera-settings controls are on the status bar and will be hidden.

`UIELEMENT_UI = 4`
This UI element indicates whether or not the camera dialog boxes can be invoked) from buttons on the status bar. If this element is disabled and the status-bar is active, the status bar will become disabled. Conversely, if the status bar is hidden (not enabled), this setting is irrelevant because the buttons will be hidden.

`UIELEMENT_CAMERA = 5`
NOT IMPLEMENTED

lFlags

NOT IMPLEMTED

lEnable

Whether or not an element is to be enabled (1) or disabled (0).

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

Typically, a developer will only enable or disable the status bar. Enabling and disabling the other UI elements is a more advanced feature.

StartVideoHook (Method)

StartVideoHook starts the video notification callback. The video notification callback (see **Notifications**), enables video frames to be transferred from the video control to the calling application. This allows applications direct access to video data in the fastest possible manner. You do not need to call this method to receive other notification events such as motion events, etc.

```
HRESULT StartVideoHook(
      long lFlag,
      long* plResult
      );
```

Parameters

lFlag

Currently, this parameter is unused and must be set to zero.

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

This is an advanced feature and should be used with care. The video notification callback allows real-time access to video data. This can be CPU intensive. Only use it when you need to catch the video frames from the camera. For example, if you are writing a motion detector and you need access to the live video stream. The **StartVideoHook** method would allow you to access real-time video data to perform a motion detection algorithm.

Example

This fragment demonstrates using the **StartVideoHook** method to receive real-time video data.

```
long lResult;
gpVideo->StartVideoHook(0,&lResult);
.
.

STDMETHOD(PortalNotification)(
        long lMsg,
        long lParam1,
        long lParam2,
        long lParam3)
{
    switch( lMsg )
    {
    case NOTIFICATIONMSG_VIDEOHOOK:
        {
            LPBITMAPINFOHEADER lpbi = (LPBITMAPINFOHEADER) lP1;
            LPBYTE lpBytes = (LPBYTE) lP2;

            unsigned long lTimeStamp = (unsigned long)lP3;
        }
    }
}
```

StopVideoHook (Method)

The **StopVideoHook** ends the video notification callback.

```
HRESULT StopVideoHook(
      long lFlag,
      long* plResult
      );
```

Parameters

lFlag

Currently, this parameter is unused and must be set to zero.

plResult

Points to a long that receives the error code of the function. Currently, there are only two error codes available. 1 = Success; 0 = Failure.

Return Values

S_OK = Success; All other values = Failure.

Remarks

Use this method to stop the video hook notification. See **StartVideoHook**.

Properties

get_CameraConnected (Property)

get_CameraConnected indicates if a camera is connected to this portal instance. The term “connection” refers to a driver connection and not a physical hardware connection.

```
HRESULT get_CameraConnected(BOOL* pConnected);
```

Parameters

pConnected

Points to a BOOL value that receives the connected state of the video portal.

Return Values

S_OK = Success; All other values = Failure.

Remarks

pConnected points to a value containing one of the following values:

1 = a camera is currently connected.
0 = no camera is connected.

get_CameraIndex (Property)

get_CameraIndex returns the index of the currently connected camera, if any.

```
HRESULT get_CameraIndex(long* plIndex);
```

Parameters

plIndex

Points to a long value that receives the index of the currently connected camera, or “-1” if no camera are currently connected.

Return Values

S_OK = Success; All other values = Failure.

Remarks

None.

get_PictureSound (Property)

get_PictureSound returns the name of the sound file played when a picture is taken.

```
HRESULT get_PictureSound( BSTR* pstrSound );
```

Parameters

pstrSound

Points to BSTR value that receives the name of the **.wav** sound file played when a picture is taken.

Return Values

S_OK = Success; All other values = Failure.

Remarks

None.

put_PictureSound (Property)

The **put_PictureSound** property sets the name of the sound file to be played when a picture is taken.

```
HRESULT put_PictureSound(BSTR strSound);
```

Parameters

strSound

BSTR value specifying the name of the **.wav** sound file to play when a picture is taken.

Return Values

S_OK = Success; All other values = Failure.

Remarks

None.

get_StampTextColor (Property)

The **get_StampTextColor** property retrieves the text color used during text stamp operations.

```
HRESULT get_StampTextColor(OLE_COLOR* pColor);
```

Parameters

pColor

Points to an OLE_COLOR that receives the color of text overlay.

Return Values

S_OK = Success; All other values = Failure.

Remarks

OLE_COLOR can be cast as an COLORREF value.
See Section 3.4 Overlay Text.

put_StampTextColor (Property)

The **put_StampTextColor** property sets the text color used during text stamp operations.

```
HRESULT put_StampTextColor(OLE_COLOR color );
```

Parameters

color

An OLE_COLOR value specifying the color of text overlay.

Return Values

S_OK = Success; All other values = Failure.

Example

The following code fragment demonstrates using **put_StampTextColor** to specify the text stamp text color.

```
COLORREF textColor = RGB(0,0,255);

gpVideo->put_StampTextColor( (OLE_COLOR)textColor );
```

The text stamp properties from the code fragment above, produces a text overlay as shown in the picture to the right. Notice that the text color is blue, i.e., 0,0,255.

get_StampFontName (Property)

The **get_StampFontName** property retrieves the name of the font used in text stamp operations.

```
HRESULT get_StampFontName( BSTR* pstrFontName );
```

Parameters

pstrFontName

Points to a BSTR value that receives the name of the font used in text overlay operations.

Return Values

S_OK = Success; All other values = Failure.

Remarks

None.

put_StampFontName (Property)

The **put_StampFontName** property sets the name of the font used in text stamp operations.

```
HRESULT put_StampFontName(BSTR strFontName );
```

Parameters

strFontName

BSTR value that specifies the name of the font to be used in text overlay operations.

Return Values

S_OK = Success; All other values = Failure.

Example

This fragment demonstrates using **put_StampFontName** to specify the text stamp font.

```
WCHAR wstrFont[] = L"Courier";
BSTR bStrFont = ::SysAllocString(bStrFont);

gpVideo->put_StampFontName(bStrFont );

gpVideo->put_StampTextColor( (OLE_COLOR) RGB(0,0,255) );
gpVideo->put_StampPointSize( 10 );
gpVideo->put_StampTextShadow( FALSE );
gpVideo->put_StampTransparentBackground( TRUE );

::SysFreeString( bStrFont );
```

The text stamp properties from the code fragment above produces a text overlay as shown to the right. Notice that the font is "Courier".

get_StampPointSize (Property)

The **get_StampPointSize** property retrieves the point-size of the font used in text stamp operations.

```
HRESULT get_StampPointSize(long* plFontSize);
```

Parameters

plFontSize

Points to a long value that receives the point-size of the font used in text overlay operations.

Return Values

S_OK = Success; All other values = Failure.

Remarks

None.

put_StampPointSize (Property)

The **put_StampPointSize** property sets the point-size of the font used in text stamp operations.

```
HRESULT put_StampPointSize(long lFontSize );
```

Parameters

lFontSize

A long value that specifies the point-size of the font to be used in text overlay operations.

Return Values

S_OK = Success; All other values = Failure.

Example

The following code fragment demonstrates using **put_StampPointSize** to specify font size.

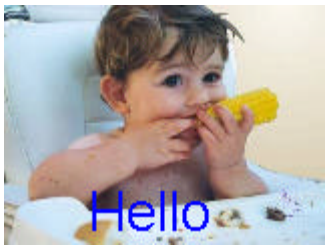
```
gpVideo->put_StampPointSize( 20 );

WCHAR wstrFont[] = L"Arial";
BSTR bStrFont = ::SysAllocString(bStrFont);

gpVideo->put_StampFontName( bStrFont );
gpVideo->put_StampTextColor( (OLE_COLOR) RGB(0,0,255) );
gpVideo->put_StampTextShadow( FALSE );
gpVideo->put_StampTransparentBackground( TRUE );

::SysFreeString( bStrFont );
```

The text stamp properties from the code fragment above, produces a text overlay as shown in the picture to the right. Notice that the font-size is large.



get_StampTextShadow (Property)

The **get_StampTextShadow** property is used to determine whether or not overlay text is shadowed.

```
HRESULT get_StampTextShadow(BOOL* pShadow);
```

Parameters

pShadow

Points to a BOOL value that indicates whether or not text overlay is shadowed.

Return Values

S_OK = Success; All other values = Failure.

Remarks

None.

put_StampTextShadow (Property)

The **put_StampTextShadow** property specifies whether or not overlay text is shadowed.

```
HRESULT put_StampTextShadow(BOOL bShadow);
```

Parameters

bShadow

A BOOL value that specifies whether or not to use a text shadow. 1 enables the text shadow; 0 disables the text shadow.

Return Values

S_OK = Success; All other values = Failure.

Remarks

To set the color of the text shadow use the following property: **put_StampTextShadowColor**.

Example

This code fragment demonstrates using **put_StampTextShadow** to specify whether or not to use a text shadow.

```
gpVideo->put_StampTextShadow( TRUE );
gpVideo->put_StampTextShadowColor( (OLE_COLOR) RGB( 0, 0, 255);

gpVideo->put_StampPointSize( 20 );

WCHAR wstrFont[] = L"Arial";
BSTR bStrFont = ::SysAllocString(bStrFont);

gpVideo->put_StampFontName( bStrFont );
gpVideo->put_StampTextColor( (OLE_COLOR) RGB(255,255,255) );
gpVideo->put_StampTextShadow( TRUE );
gpVideo->put_StampTransparentBackground( TRUE );

::SysFreeString( bStrFont );
```

The text stamp properties from the code fragment above, produces a text overlay as shown in the picture to the right. Notice that the text overlay contains a blue shadow.

get_StampTextShadowColor (Property)

The **get_StampTextShadowColor** property retrieves the color of the overlay text shadow.

```
HRESULT get_StampTextShadowColor(OLE_COLOR* pShadowColor);
```

Parameters

pShadowColor

Points to an OLE_COLOR value that indicates the color of shadowed text.

Return Values

S_OK = Success; All other values = Failure.

Remarks

OLE_COLOR can be cast as an COLORREF value.
See Section 3.4 Overlay Text.

put_StampTextShadowColor (Property)

The **put_StampTextShadowColor** property sets the color of the shadow (if any) for overlay text.

```
HRESULT put_StampTextShadowColor(OLE_COLOR textShadowColor );
```

Parameters

textShadowColor

An OLE_COLOR value that specifies the color of shadowed text.

Return Values

S_OK = Success; All other values = Failure.

Example

The following code fragment demonstrates using **put_StampTextShadowColor** to specify the color of the text shadow.

```
gpVideo->put_StampTextShadowColor( RGB( 0, 0, 255);
gpVideo->put_StampTextShadow( TRUE );

gpVideo->put_StampPointSize( 20 );

WCHAR wstrFont[] = L"Arial";
BSTR bStrFont = ::SysAllocString(bStrFont);

gpVideo->put_StampFontName( bStrFont );
gpVideo->put_StampTextColor( (OLE_COLOR)RGB(255,255,255) );
gpVideo->put_StampTextShadow( TRUE );
gpVideo->put_StampTransparentBackground( TRUE );

::SysFreeString( bStrFont );
```

The text stamp properties from the code fragment above, produces a text overlay as shown in the picture to the right. Notice that the text overlay contains a blue shadow.

get_StampTransparentBackground (Property)

The **get_StampTransparentBackground** property is used to determine whether or not overlaid text is transparent or opaque during text stamp operations.

```
HRESULT get_StampTransparentBackground( BOOL* pTransparent);
```

Parameters

pTransparent

Points to a BOOL value indicating whether or not the background of overlaid text is transparent or opaque.

Return Values

S_OK = Success; All other values = Failure.

Remarks

pTransparent points to a BOOL which indicates whether or not overlaid text is transparent or opaque. The BOOL pTransparent points to is 1 if transparent, and 0 if opaque. When the background is opaque, its color is determined by **put_StampBackgroundColor**.

put_StampTransparentBackground (Property)

The **put_StampTransparentBackground** property specifies whether or not overlaid text is transparent or opaque during text stamp operations.

```
HRESULT put_StampTransparentBackground(BOOL bTransparent);
```

Parameters

bTransparent

A BOOL value specifying whether or not the background of overlaid text is transparent or opaque. 1 (TRUE) indicates a transparent background; 0 (FALSE) indicates an opaque background

Return Values

S_OK = Success; All other values = Failure.

Example

The following code fragment demonstrates using **put_StampTransparentBackground** to specify whether or not the text stamp is opaque or transparent. This example specifies the text stamp to be opaque.

```
gpVideo->put_StampTransparentBackground( FALSE );

gpVideo->put_StampBackgroundColor( (OLE_COLOR) RGB(0,0,255);
gpVideo->put_StampTextShadow( FALSE );
gpVideo->put_StampPointSize( 20 );

WCHAR wstrFont[] = L"Arial";
BSTR bStrFont = ::SysAllocString(bStrFont);

gpVideo->put_StampFontName( bStrFont );
gpVideo->put_StampTextColor( (OLE_COLOR) RGB(255,255,255) );
gpVideo->put_StampTextShadow( TRUE );

::SysFreeString( bStrFont );
```

The text stamp properties from the code fragment above, produces a text overlay as shown in the picture to the right. Notice that the text overlay is not transparent and the background color is blue.

get_StampBackgroundColor (Property)

The **get_StampBackgroundColor** property retrieves the color of the text overlay background.

```
HRESULT get_StampBackgroundColor(OLE_COLOR* pBackgroundColor);
```

Parameters

pBackgroundColor

Points to an OLE_COLOR value that indicates the background color of an opaque background used during text stamp operations.

Return Values

S_OK = Success; All other values = Failure.

Remarks

The property is only used if the overlay text background is not transparent. See **put_StampTransparentBackground**

OLE_COLOR can be cast as an COLORREF value.
See Section 3.4 Overlay Text.

put_StampBackgroundColor (Property)

The **put_StampBackgroundColor** property sets the background color used during overlay text stamp operations.

```
HRESULT put_StampBackgroundColor(OLE_COLOR backgroundColor);
```

Parameters

backgroundColor

An OLE_COLOR that specifies the background color of an opaque background used during text stamp operations.

Return Values

S_OK = Success; All other values = Failure.

Remarks

This color is only used if the overlay text background is not transparent. See **put_StampTransparentBackground**

Example

The following code fragment demonstrates using the put_StampBackgroundColor method to specify the color of an opaque background.

```
gpVideo->put_StampBackgroundColor( (OLE_COLOR)RGB(255,0,0);
gpVideo->put_StampTransparentBackground( FALSE );

m_cVideo.put_StampBackgroundColor( RGB(0,0,255);

gpVideo->put_StampTextShadow( FALSE );
gpVideo->put_StampPointSize( 20 );

WCHAR wstrFont[] = L"Arial";
BSTR bStrFont = ::SysAllocString(bStrFont);

gpVideo->put_StampFontName( bStrFont );
gpVideo->put_StampTextColor( (OLE_COLOR) RGB(255,255,255) );
gpVideo->put_StampTextShadow( TRUE );

::SysFreeString( bStrFont );
```

The text stamp properties from the code fragment above, produces a text overlay as shown in the picture to the right. Notice that the text overlay is not transparent and the background color is red.

get_EnablePreview (Property)

The **get_EnablePreview** property indicates whether or not video preview is active.

```
HRESULT get_EnablePreview(BOOL* pPreview);
```

Parameters

pPreview

Points to a BOOL value indicating whether or not video preview is active.

Return Values

S_OK = Success; All other values = Failure.

Remarks

1 = Video preview is enabled
0 = Video preview is disabled

put_EnablePreview (Property)

The **put_EnablePreview** property enables or disables video preview.

```
HRESULT put_EnablePreview(BOOL bPreview);
```

Parameters

bPreview

A BOOL value specifying whether or not to enable/disable video preview.

Return Values

S_OK = Success; All other values = Failure.

Remarks

By default the preview is not enabled. You must enable the preview before you can see video.

get_MovieVideoCompressionFOURCC (Property)

The **get_MovieVideoCompressionFOURCC** property retrieves the current four-character code of the VFW installable compressor used during movie recording.

```
HRESULT get_MovieVideoCompressionFOURCC(long* plFourCC);
```

Parameters

plFourCC

Points to a long value indicating the four-character code of the VFW installable compressor used during movie recording.

Return Values

S_OK = Success; All other values = Failure.

Remarks

None.

put_MovieVideoCompressionFOURCC (Property)

The **put_MovieVideoCompressionFOURCC** property selects the video codec to use for recording.

```
HRESULT put_MovieVideoCompressionFOURCC(long lFourCC);
```

Parameters

lFourCC

A long value indicating the four-character code of the VFW installable compressor to use during movie recording.

Return Values

S_OK = Success; All other values = Failure.

Remarks

This property uses a four-character code to specify the VFW installable compressor to be used during movie recording. To write raw RGB frames to disk, specify 0 (zero) for the compression parameter. If you choose an installable compressor that cannot keep up with the desired frame rate, the resulting video will be poor quality.

You can use the mmioFOURCC macro defined in "mmsystem.h" to specify an installable compressor to use during video compression. For example the Intel Indeo 5.2 codec has a four-character code, which when used with mmioFOURCC appears as follows:

```
long lCompressor = mmioFOURCC('I','V','5','0')
```

The default compressor is Indeo 5.2 with factory compression settings.

Examples

This code fragment selects the Intel Indeo 5.2 installable compressor.

```
long lCompressor = 0x30355649;
gpVideo->put_MovieVideoCompressionFOURCC( lCompressor );
```

This code fragment specifies using no video compressor, which writes raw RGB frames to disk.

```
gpVideo->put_MovieVideoCompressionFOURCC( 0 );
```

get_MovieVideoCompressionKeyFrameInterval (Property)

get_MovieVideoCompressionKeyFrameInterval property retrieves the key frame interval rate used during movie recording.

```
HRESULT get_MovieVideoCompressionKeyFrameInterval(long *plKeyInterval);
```

Parameters

plKeyInterval

Points to a long value indicating the key-frame interval used during movie recording.

Return Values

S_OK = Success; All other values = Failure.

Remarks

Retrieves the rate at which key frames are generated when recording a video. For example, 15 indicates that a key frame is generated for every 15 frames recorded. Normally, this property is not used.

put_MovieVideoCompressionKeyFrameInterval (Property)

The **put_MovieVideoCompressionKeyFrameInterval** property sets the key frame interval rate used during movie recording.

```
HRESULT put_MovieVideoCompressionKeyFrameInterval(long lKeyInterval);
```

Parameters

lKeyInterval

A long value specifying the key-frame interval used during movie recording.

Return Values

S_OK = Success; All other values = Failure.

Remarks

Setting this property sets the rate at which key frames are generated when recording a video. For example, 15 indicates every 15 frames, 1 key frame is generated.

get_MovieVideoCompressionQuality (Property)

The **get_MovieVideoCompressionQuality** property retrieves the compression quality setting used during movie recording.

```
HRESULT get_MovieVideoCompressionQuality(long* plQuality);
```

Parameters

plQuality

Points to a long value which indicates the compression quality setting used during movie recording. This value ranges from 0..10000.

Return Values

S_OK = Success; All other values = Failure.

Remarks

None.

put_MovieVideoCompressionQuality (Property)

The **put_MovieVideoCompressionQuality** property sets the compression quality setting used during movie recording.

```
HRESULT put_MovieVideoCompressionQuality(long lQuality);
```

Parameters

lQuality

A long value specifying the compression quality used during movie recording. This value ranges from 0..10000. If –1 is specified, the default compression setting is used.

Return Values

S_OK = Success; All other values = Failure.

Remarks

None.

get_MoviePlaybackFPS (Property)

The **get_MoviePlaybackFPS** property retrieves the FPS (frames per second) rate at which a movie recorded with the video control will play back at.

```
HRESULT get_MoviePlaybackFPS(long* plFPS);
```

Parameters

plFPS

Points to a long value indicating the frames per second that a recorded video will play back.

Return Values

S_OK = Success; All other values = Failure.

Remarks

This property retrieves the FPS in which a recorded movie would playback. For example, if you record a movie @ 30fps, it would naturally playback at 30fps. However, imagine recording an animation movie, which you manually add frames to an AVI file. In this situation, are able to specify what the playback rate of the animation would be. Thus, if you added a frame once every hour, you could specify a playback rate of 15 FPS (frames per second) the video would play-back 15 hours worth of video every second.

put_MoviePlaybackFPS (Property)

The **put_MoviePlaybackFPS** property sets the FPS (frames per second) rate at which a movie recorded with the video control will play back at.

```
HRESULT put_MoviePlaybackFPS(long lFPS);
```

Parameters

lFPS

A long value specifying the number of frames per second that a recorded video will play back.

Return Values

S_OK = Success; All other values = Failure.

Remarks

Use this property to set the rate, in frames per second, at which a subsequent video will be recorded.

Exception: For single-frame (time-lapse or animation) movies, this setting has no effect on recording but *still determines the playback rate* of the resulting movie when it is played later.

The following table details the currently supported cameras and their frame per second limit.

Camera	160x120	320x240	640x480
Logitech QuickCam Pro	30 FPS	30 FPS	3[2]
Logitech QuickCam VC	30 FPS	15 FPS	3[2]
Logitech QuickCam Express	30 FPS	15 FPS	N/A
Logitech QuickCam Home	30 FPS	25 FPS[1]	3[2]

1. The Logitech QuickCam Home supports a quality compression setting. When this setting is *optimize-speed* the maximum frames per second is 25. When set to *optimize-quality* the maximum frames per second is 15.

2. For cameras which support 640x480 mode, the maximum FPS depends on the speed in which video frames can be written to disk. For faster machines, e.g.,. in excess of 333MHZ, the maximum FPS is 7. For slower machines it is 3 FPS.

This property specifies the FPS in which a recorded movie would playback. For example, if you record a movie @ 30fps, it would naturally playback at 30fps. However, imagine recording an animation movie, which you manually add frames to an AVI file. In this situation, are able to specify what the playback rate of the animation would be. Thus, if you added a frame once every hour, you could specify a playback rate of 15 FPS (frames per second) the video would play-back 15 hours worth of video every second.

get_MovieAudioSamplesPerSecond (Property)

The **get_MovieAudioSamplesPerSecond** property retrieves the audio sampling rate for movie recording. Currently, this method is not implemented.

```
HRESULT get_MovieAudioSamplesPerSecond(long* plAudioSampleRate);
```

Parameters

plAudioSampleRate

Points to a long value which receives the audio sampling rate used during movie recording.

Remarks

NOT IMPLEMENTED, audio is currently recorded at 11,025 samples per second.

put_MovieAudioSamplesPerSecond (Property)

The **put_MovieAudioSamplesPerSecond** property sets the audio sampling rate for movie recording.

```
HRESULT put_MovieAudioSamplesPerSecond(long lSamples);
```

Parameters

lSamples

A long value indicating the audio sampling rate used during movie recording.

Remarks

NOT IMPLEMENTED, audio is currently recorded at 11,025 samples per second.

get_MovieAudioChannels (Property)

The **get_MovieAudioChannels** property retrieves the number of audio channels used during movie recording. This method is not currently implemented.

```
HRESULT get_MovieAudioChannels(long *plChannels)
```

Parameters

plSamples

Points to a long value that receives the number of audio channels used during movie recording.

Remarks

NOT IMPLEMENTED, audio is recorded with one audio channel (mono).

put_MovieAudioChannels (Property)

The **put_MovieAudioChannels** property sets the number of audio channels used during movie recording. This method is not currently implemented.

```
HRESULT put_MovieAudioChannels(long lChannels);
```

Parameters

lChannels

A long value that indicates the number of audio channels used during movie recording.

Remarks

NOT IMPLEMENTED, audio is recorded with one audio channel (mono).

get_MovieAudioBitsPerSample (Property)

get_MovieAudioBitsPerSample retrieves the bits per sample used for audio recording. This method is not currently implemented.

```
HRESULT get_MovieAudioBitsPerSample(long* plSampleSize);
```

Parameters

plSampleSize

Points to a long value that receives the number of audio channels used during movie recording.

Remarks

NOT IMPLEMENTED, audio is recorded at 8 bits per sample.

put_MovieAudioBitsPerSample (Property)

The **put_MovieAudioBitsPerSample** property sets the bits per audio sample used for movie recording. This method is not currently implemented.

```
HRESULT put_MovieAudioBitsPerSample(long lSampleSize);
```

Parameters

lSampleSize

A long value that specifies the number of bits per audio sample used during movie recording.

Remarks

NOT IMPLEMENTED, audio is recorded at 8 bits per sample.

get_MovieAudioCompressionFOURCC (Property)

This method is not currently implemented.

```
HRESULT get_MovieAudioCompressionFOURCC(long* plCompressorFourCC);
```

Remarks

NOT IMPLEMENTED, audio is not compressed by default.

put_MovieAudioCompressionFOURCC (Property)

This method is not currently implemented.

```
HRESULT put_MovieAudioCompressionFOURCC(long lCompressorFourCC);
```

Remarks

NOT IMPLEMENTED, audio is not compressed by default.

get_MovieAudioCompressionQuality (Property)

This method is not currently implemented.

```
HRESULT get_MovieAudioCompressionQuality(long* plQuality);
```

Remarks

NOT IMPLEMENTED, audio is not compressed by default.

put_MovieAudioCompressionQuality (Property)

This method is not currently implemented.

```
HRESULT put_MovieAudioCompressionQuality(long lQuality);
```

Remarks

NOT IMPLEMENTED, audio is not compressed by default.

get_MovieRecordAudio (Property)

The **get_MovieRecordAudio** property determines whether or not audio will be recorded during movie capture.

```
HRESULT get_MovieRecordAudio(BOOL * pbRecordAudio );
```

Parameters

pbRecordAudio

Points to a BOOL value indicating whether or not audio is recorded during movie capture.

Return Values

S_OK = Success; All other values = Failure.

Remarks

None.

put_MovieRecordAudio (Property)

The **put_MovieRecordAudio** property specifies whether or not audio is recorded during movie capture.

```
HRESULT put_MovieRecordAudio(BOOL bRecordAudio);
```

Parameters

bRecordAudio

A BOOL value specifying whether or not to record audio during movie capture.

Return Values

S_OK = Success; All other values = Failure.

Remarks

0 means ‘do not record audio’. Any other value causes audio to be recorded during movie recording. Audio recording is *on* by default.

get_MovieRecordMode (Property)

The **get_MovieRecordMode** property determines the mode used during movie capture. The "movie mode" used determines how the movie (AVI file) is created and written to disk.

```
HRESULT get_MovieRecordMode(long* plMovieMode );
```

Parameters

plMovieMode

Points to a long value indicating the "movie mode" used during movie capture.

Return Values

S_OK = Success; All other values = Failure.

Remarks

The following movie modes are defined and can be found in LVServerDefs.H header file:

SEQUENCECAPTURE_FPS_USERSPECIFIED = 1,
SEQUENCECAPTURE_FPS_FASTASPOSSIBLE = 2,
STEPCAPTURE_MANUALTRIGGERED = 3,

See **put_MovieRecordMode** for more details.

put_MovieRecordMode (Property)

The **put_MovieRecordMode** property specifies the mode used during movie capture. The "movie mode" used determines how the movie (AVI file) is created and written to disk.

```
HRESULT put_MovieRecordMode(long lMovieMode);
```

Parameters

lMovieMode

A long value specifying the "movie mode" used during movie capture.

Return Values

S_OK = Success; All other values = Failure.

Remarks

Use this property to select the movie record mode used while recording movies. The following modes are currently defined and can be found in LVServerDefs.H header file:

SEQUENCECAPTURE_FPS_FASTASPOSSIBLE

This is the default recording mode. It indicates that video should be recorded at the maximum frame rate supported by the camera and image size.

SEQUENCECAPTURE_FPS_USERSPECIFIED

This recording mode is used when the frame rate written to disk is to be specified by the user. See **put_MoviePlaybackFPS** for related information.

STEPCAPTURE_MANUALTRIGGERED

This recording mode is used for both time-lapse and stop-motion movies. In this mode you open the movie as usual with **StartMovieRecording**, then use **StepCaptureAddFrame** to add each frame. You can use **put_MoviePlaybackFPS** to specify the frame rate at which the movie is to play back. See Section 3.6 for example code.

get_MovieCreateFlags (Property)

The **get_MovieCreateFlags** property retrieves the movie creation flags for movies (AVI files).

```
HRESULT get_MovieCreateFlags(long* plCreateFlags );
```

Parameters

plCreateFlags

Points to a long value indicating creation flags used when creating a movie.

Return Values

S_OK = Success; All other values = Failure.

Remarks

The following creation flags are defined and can be found in LVServerDefs.H header file. See **put_MovieCreateFlags** for more information.

MOVIECREATEFLAGS_CREATENEW	1
MOVIECREATEFLAGS_APPEND	2

put_MovieCreateFlags (Property)

put_MovieCreateFlags sets the creation flags for AVI files.

```
HRESULT put_MovieCreateFlags( long lCreateFlags);
```

Parameters

lCreateFlags

A long value indicating creation flags used when creating a movie.

Return Values

S_OK = Success; All other values = Failure.

Remarks

This property determines what happens when the portal attempts to create an AVI file and there is an existing file of the same name.

MOVIECREATEFLAGS_CREATENEW always creates an AVI file, overwriting an existing file. MOVIECREATEFLAGS_APPEND appends to an existing AVI file, or creates a new AVI file if there is no file to append to.

The following creation flags are defined and can be found in LVServerDefs.H header file.

MOVIECREATEFLAGS_CREATENEW	1
MOVIECREATEFLAGS_APPEND	2

get_MovieRecordingActiveLocal (Property)

The **get_MovieRecordingActiveLocal** property determines if “this” video control is currently recording a movie.

```
HRESULT get_MovieRecordingActiveLocal(BOOL* plRecordActive );
```

Parameters

plRecordActive

Points to a BOOL value indicating whether or not a movie is currently being recorded by “this” portal instance.

Return Values

S_OK = Success; All other values = Failure.

Remarks

Only one Video Portal instance can record a movie at a time. This method allows you to determine if *this particular instance* of Video Portal is recording. If recording is going on through another video portal, this property will still return 0.

See the **get_MovieRecordingActiveGlobal** property to find out if any video portal on the host system (including this one) is recording.

get_MovieRecordingActiveGlobal (Property)

The **get_MovieRecordingActiveGlobal** property determines if a movie is being recorded by any video portal instance.

```
HRESULT get_MovieRecordingActiveGlobal(BOOL* plActiveGlobal);
```

Parameters

plActiveGlobal

Points to a BOOL value indicating whether or not a movie is currently being recorded by “any” portal instance.

Return Values

S_OK = Success; All other values = Failure.

Remarks

Another portal instance could be currently recording a movie. The Video Portal only allows one instance to record a movie at a time. This method allows you to determine if any instance of the portal on the host system is currently recording a movie.

get_CameraState (Property)

The **get_CameraState** property determines the current state of the connected camera.

```
HRESULT get_CameraState(long *plState );
```

Parameters

plState

Points to a long value indicating the current state of the connected camera.

Return Values

S_OK = Success; All other values = Failure.

Remarks

The video portal monitors the state of the camera at all times. Should the camera be unplugged from the USB port, a notification event informs the application of the new unplugged status. Additionally, using this method can also be helpful in determining your application's course of action.

The camera state can be any one of the following values, which can be found in the LVServerDefs.H header file:

CAMERA_OK

Indicates that everything is normal with the camera.

CAMERA_UNPLUGGED

Indicates that the camera has been unplugged from the USB port. When the camera is plugged back into the computer, camera state will change to CAMERA_OK.

CAMERA_INUSE

Indicates that another application is using the camera. Camera applications, which do not use the QuickCam SDK, cause this camera state to occur. However, the QuickCam SDK will soon provide for mechanisms that allow even non-QuickCam SDK applications to share the camera.

CAMERA_ERROR

Indicates that a hardware or driver related problem has occurred.

CAMERA_SUSPENDED

Indicates that the computer has entered suspend mode. When the computer resumes from suspend mode, the camera state will be correctly updated.

CAMERA_UNKNOWNSTATUS

This state is reserved.

get_EnableMovieRecordErrorPrompt (Property)

The **get_EnableMovieRecordErrorPrompt** property determines if error prompting is enabled during movie capture.

```
HRESULT get_EnableMovieRecordErrorPrompt(BOOL* pErrorPrompt);
```

Parameters

pErrorPrompt

Points to a BOOL value indicating whether or not movie error message prompting is enabled.

Return Values

S_OK = Success; All other values = Failure.

Remarks

When error prompting is enabled for movie recording, message boxes appear from the video portal window describing the error. See the reference section on **put_EnableMovieRecordErrorPrompt** for more information.

put_EnableMovieRecordErrorPrompt (Property)

The **put_EnableMovieRecordErrorPrompt** property enables or disables error prompting during movie capture.

```
HRESULT put_EnableMovieRecordErrorPrompt(BOOL bErrorPrompt);
```

Parameters

bErrorPrompt

A BOOL value specifying whether or not to enable movie error message prompting.

Return Values

S_OK = Success; All other values = Failure.

Remarks

When error prompting is enabled for movie recording and an error occurs, the following error strings are formatted in a message box. The string fitting the cause of the error is chosen.

Unable to open an audio device for recording. Either another application is using the audio device for recording or the audio device is not working properly.
Another application is recording audio. Stop recording with this other application and then try to record another video.
Unable to open the audio device for recording. The audio device is not installed or is not working properly.
Unable to save video to %s. The "%s" device has been removed from the system. To save a video, plug the camera back into the USB port.
Unable to save video to "%s". Another activity is currently recording a video.
Unable to save video to %s. The camera device is still waking up from suspend mode.
Unable to save video to %s. The camera device has been removed from the system. To save a video, plug the camera back into the USB port.
Unable to save video to "%s". The "%s" camera device is currently in use by another application.
Unable to save video to "%s". The camera device is currently in use by another application.
Unable to save video to "%s". The camera device was not found or is not working properly. Check to make sure the camera is properly plugged into the USB port.
Unable to save video to "%s". Another activity is currently recording a video.
Unable to save video "%s". The specified file is open and cannot be overwritten.
Unable to save video to "%s".
Unable to save video to "%s". Drive %s is write protected. Remove the write-protection before saving another video.
Unable to save video to "%s". Drive %s is not accessible or is not ready.
Unable to save video to "%s". There are only %s bytes available on drive %s. Additional disk space is

needed to save a video.
Unable to save video "%s". The specified file is open and cannot be overwritten.
Unable to save video to "%s". Unable to find the specified path.
Unable to save video to "%s". The specified path and drive are inaccessible.

get_EnablePictureDiskErrorPrompt (Property)

The **get_EnablePictureDiskErrorPrompt** property determines if error prompting is enabled during picture to disk operations.

```
HRESULT get_EnablePictureDiskErrorPrompt(BOOL *pErrorPrompt);
```

Parameters

pErrorPrompt

Points to a BOOL value indicating whether or not picture error message prompting is enabled.

Return Values

S_OK = Success; All other values = Failure.

Remarks

See the reference section on **put_EnablePictureDiskErrorPrompt** for more information.

put_EnablePictureDiskErrorPrompt (Property)

The **put_EnablePictureDiskErrorPrompt** property is used to enable/disable error prompting during picture to disk operations.

```
HRESULT put_EnablePictureDiskErrorPrompt(BOOL bErrorPrompt);
```

Parameters

bErrorPrompt

A BOOL value specifying whether or not picture error message prompting is enabled.

Return Values

S_OK = Success; All other values = Failure.

Remarks

When error prompting is enabled for the picture to disk operation and an error occurs, one of the following error strings is formatted in a message box. The string fitting the cause of the error is chosen

Unable to save picture to “%s”. The "%s" device has been removed from the system. To take a picture, plug the camera back into the USB port.
Unable to save picture to "%s". The specified path and drive are inaccessible.
Unable to save picture to "%s". Drive %s is write protected. Remove the write-protection before saving another picture.
Unable to save picture to "%s". Drive %s is not accessible or is not ready.
Unable to save picture to "%s". There are only %s bytes available on drive %s. Additional disk space is needed to save a picture.
Unable to save picture to "%s". Unable to find the specified path.
Unable to save video to "%s". Drive %s is write protected. Remove the write-protection before saving another video.
Unable to save picture to "%s". The specified file already exists and is marked as read-only.
Unable to save picture to "%s". The specified file name contains an invalid extension.
Unable to save picture to "%s".
Unable to save picture to "%s". The camera device was not found or is not working properly. Check to make sure the camera is properly plugged into the USB port.
Unable to save picture to "%s". The "%s" camera device is currently in use by another application.
Unable to save picture to %s. The camera device has been removed from the system. To take a picture, plug the camera back into the USB port.
Unable to save picture to "%s". The camera device is currently in use by another application.
Unable to save picture to “%s”. The camera device is still waking up from suspend mode.

put_StatusBarText (Property)

The **put_StatusBarText** property specifies text to be displayed in the status bar of the Video Portal window. This method is not currently implemented.

```
HRESULT put_StatusBarText(BSTR strText );
```

Remarks

NOT CURRENTLY IMPLEMENTED

get_PreviewMaxWidth (Property)

The **get_PreviewMaxWidth** property retrieves the current maximum preview display size of the video portal window.

```
HRESULT get_PreviewMaxWidth(long *plMaxWidth );
```

Parameters

plMaxWidth

Points to a long value indicating the maximum preview display width of the video portal window.

Return Values

S_OK = Success; All other values = Failure.

Remarks

The default and recommended preview size is 320x240. However, you can change the width to fit your application needs. The streaming video preview will be scaled to fit within this size.

put_PreviewMaxWidth (Property)

The **put_PreviewMaxWidth** property sets the maximum width, in pixels, of the video preview window.

```
HRESULT put_PreviewMaxWidth(long lMaxWidth);
```

Parameters

lMaxWidth

A long value specifying the maximum preview display width of the video portal window.

Return Values

S_OK = Success; All other values = Failure.

Remarks

The default and recommended preview size is 320x240. However, you can change the width to fit your application needs. The streaming video preview will be scaled to fit within this size.

Note that all conventional video formats have a width to height ratio of 4:3, so to avoid distortion of the preview video, you are advised to set the PreviewMaxWidth and PreviewMaxHeight in the same 4:3 ratio.

Example

The following code fragment demonstrates how to change the preview width and height of the video preview area.

```
gpVideo->put_PreviewMaxHeight( 480 );
gpVideo->put_PreviewMaxWidth( 640 );
```

get_PreviewMaxHeight (Property)

The **get_PreviewMaxHeight** property retrieves the current maximum preview display size of the video portal window.

```
HRESULT get_PreviewMaxHeight(long *plMaxHeight );
```

Parameters

plMaxHeight

Points to a long value indicating the maximum preview display height of the video portal window.

Return Values

S_OK = Success; All other values = Failure.

Remarks

Returns a long, which specifies the current preview window height.

The default and recommended preview size is 320x240. However, you can change the height to fit your application needs. The streaming video preview will be scaled to fit within this size.

put_PreviewMaxHeight (Property)

The **put_PreviewMaxHeight** property sets the maximum height, in pixels, of the video preview window – not counting the status bar.

```
HRESULT put_PreviewMaxHeight(long lMaxHeight);
```

Parameters

lMaxHeight

A long value specifying the maximum preview display height of the video portal window.

Return Values

S_OK = Success; All other values = Failure.

Remarks

The default and recommended preview size is 320x240. However, you can change the height to fit your application needs. The streaming video preview will be scaled to fit within this size.

Note that this maximum is for just the actual preview pane, and does not include the status bar if enabled. Note also that all conventional video formats have a width to height ratio of 4:3, so to avoid distortion of the preview, you should set PreviewMaxWidth and PreviewMaxHeight in the same 4:3 ratio.

Example

The following code fragment demonstrates how to change the preview width and height of the video preview area. By default the video portals maximum preview height is 240, and the default maximum width is 320.

Notifications

Handling Notification Events

The video portal sends notification events to send information to the calling application. The video portal contains a single notification method for handling all event messages. To add the event handler to your project follow these steps:

1. Follow the instructions in section **3.2 Getting Started**. This section describes adding the event handler to your project.
2. The **PortalNotification** method described in section 3.2 is the notification callback to handle event messages from the video portal control.

The event handler referenced in section **3.2 Getting Started.**

```
STDMETHOD(PortalNotification)(
        long lMsg,
        long lParam1,
        long lParam2,
        long lParam3)
{
    // implement any PortalNotification notification handling here.
    . . .
}
```

The first parameter **lMsg** indicates the ID of the notification event. The remaining parameters contain notification specific information. The notification constants are listed below. Additionally, these events definitions can be found in the LVServerDefs.H header file.

NOTIFICATIONMSG_MOTION
NOTIFICATIONMSG_MOVIERECORDERROR
NOTIFICATIONMSG_CAMERADETACHED
NOTIFICATIONMSG_CAMERAREATTACHED
NOTIFICATIONMSG_IMAGESIZECHANGE
NOTIFICATIONMSG_CAMERAPRECHANGE
NOTIFICATIONMSG_CAMERACHANGEFAILED
NOTIFICATIONMSG_POSTCAMERACHANGED
NOTIFICATIONMSG_CAMERBUTTONCLICKED
NOTIFICATIONMSG_VIDEOHOOK
NOTIFICATIONMSG_SETTINGDLGCLOSED
NOTIFICATIONMSG_QUERYPRECAMERAMODIFICATION
NOTIFICATIONMSG_MOVIESIZE

NOTIFICATIONMSG_MOTION
This notification message is received during motion detection monitoring. To activate the motion detector, use the **SetCameraPropertyLong** method (see SetCameraPropertyLong). The only parameter used in the PortalNotification event handler for NOTIFICATIONMSG_MOTION is **lParam1**. This parameter contains the percentage of movement within the current video frame as compared to previous frames.

NOTIFICATIONMSG_MOVIERECORDERROR
This notification message is received when an error occurs during movie a recording. The only parameter used in the PortalNotification event handler for NOTIFICATIONMSG_MOVIERECORDERROR is **lParam1**. This parameter specifies the error, which occurred while recording the movie.

The following errors are possible:

WRITEFAILURE_RECORDINGSTOPPED
Indicates general movie recording error caused by insufficient disk space or disk access. Additionally, this error indicates that the AVI file contains valid information but was prematurely stopped.

WRITEFAILURE_RECORDINGSTOPPED_FILECORRUPTANDDELETED

Indicates general movie recording error caused by insufficient disk space or disk access. Additionally, this error indicates the AVI file did not close properly and had to be deleted.

WRITEFAILURE_CAMERA_SUSPENDED
Indicates movie recording was unable to continue due to the computer entering suspend mode.

WRITEFAILURE_CAMERA_UNPLUGGED
Indicates movie recording was unable to continue due to the camera being unplugged from the USB port.

NOTIFICATIONMSG_CAMERADETACHED
This notification message is received the currently connected camera device is removed from the USB port.

NOTIFICATIONMSG_CAMERAREATTACHED
This notification message is received a camera device is plugged into the computers USB port. This message is only fired if the video portal is not currently connected to a camera.

NOTIFICATIONMSG_IMAGESIZECHANGE
This notification message is received when the image size is changed from the video controls user-interface controls. The **lParam1** parameter contains the width of the new image size and **lParam2** contains the height of the new image size.

NOTIFICATIONMSG_CAMERAPRECHANGE
<NOT USED>

NOTIFICATIONMSG_CAMERACHANGEFAILED
This notification message is received after an attempt to switch from one camera connection to another fails. The index of the camera is passed in the **lParam1** parameter.

NOTIFICATIONMSG_POSTCAMERACHANGED
This notification message is received after a switch from one camera connection to another is successful. The index of the new camera is passed in the **lParam1** parameter.

NOTIFICATIONMSG_CAMERBUTTONCLICKED
This notification message is received after the camera button (on supported cameras) is pressed.

NOTIFICATIONMSG_VIDEOHOOK
This notification message is received during video streaming. Video streaming allows direct access to camera data through this notification message. To enable video streaming use the **StartVideoHook** method.

lParam1
This parameter is a pointer to a BITMAPINFOHEADER structure

lParam2
This parameter is a pointer to the actual bytes of the bitmap data.

lParam3
This parameter contains the system time stamp of the video frame in milliseconds. The system time is the time elapsed since Windows was started.

NOTIFICATIONMSG_SETTINGDLGCLOSED
<NOT USED>

NOTIFICATIONMSG_QUERYPRECAMERAMODIFICATION
<NOT USED>

NOTIFICATIONMSG_MOVIESIZE
This notification message is received during movie recording to indicate the current size in bytes of the recording video. The **lParam3** contains the current size in bytes of the movie being written to disk.

Camera Properties

Camera Properties

The QuickCam SDK provides mechanisms to control camera properties. Each camera supported by the QuickCam SDK has various properties. Some of the camera properties are available in all Logitech cameras, while others are not. **Appendix A** contains a matrix indicating camera properties and the cameras which support them. All of these camera properties are accessed by the **SetCameraPropertyLong** and **GetCameraPropertyLong** methods.

- **PROPERTY_ORIENTATION**
  This property controls the orientation of the image. It can be any of the following values:
  - **ORIENTATIONMODE_NORMAL**
    Indicates normal orientation
  - **ORIENTATIONMODE_MIRRORED**
    Indicates a mirrored orientation
  - **ORIENTATIONMODE_FLIPPED**
    Indicates a flipped orientation
  - **ORIENTATIONMODE_FLIPPED_AND_MIRRORED**
    Indicates a flipped and mirrored orientation
- **PROPERTY_BRIGHTNESSMODE**
  This property controls auto-brightness mode. When this property is set to ADJUSTMENT_AUTOMATIC, auto-brightness is engaged, otherwise auto-brightness is turned off. This property can be either ADJUSTMENT_MANUAL or ADJUSTMENT_AUTOMATIC.
- **PROPERTY_BRIGHTNESS**
  This property is used to adjust brightness and is only available when auto-brightness is turned off (see PROPERTY_BRIGHTNESSMODE). This property can range between 0..255.
- **PROPERTY_CONTRAST**
  This property is used to adjust contrast. This property can range between 0..255.
- **PROPERTY_COLORMODE**
  This property controls auto-color mode. When this property is set to ADJUSTMENT_AUTOMATIC, auto-color is engaged; otherwise auto-color mode is turned off. This property can be either ADJUSTMENT_MANUAL or ADJUSTMENT_AUTOMATIC.
- **PROPERTY_REDGAIN**
  This property adjusts red saturation and is only available when auto-color is turned off (see PROPERTY_COLORMODE). This property can range between 0..255.
- **PROPERTY_BLUEGAIN**
  This property adjusts red saturation and is only available when auto-color is turned off (see PROPERTY_COLORMODE). This property can range between 0..255.
- **PROPERTY_SATURATION**
  This property adjusts saturation. This property can range between 0..255.
- **PROPERTY_EXPOSURE**
  This property adjusts exposure. This property can range between 0..255.
- **PROPERTY_RESET**
  This property is can only be set. It is used to reset a camera to its hardware defaults.
- **PROPERTY_ANTIBLOOM**
  This property enables or disables anti-bloom. Blooming occurs in some Logitech cameras when bright lights appear to cover the entire video image. This property can be either TRUE or FALSE.
- **PROPERTY_LOWLIGHTFILTER**

This property enables or disables low-light filtering within the image. This property can be either TRUE or FALSE.

- **PROPERTY_HUE**
  This property adjusts hue. This property can range between 0..255.

- **PROPERTY_PORT_TYPE**
  This read-only property is used to retrieve the type of camera port connection.

- **PROPERTY_PICTSMART_MODE**
  This property enables or disables PictureSmart™ mode. This property can be either TRUE or FALSE.

- **PROPERTY_PICTSMART_LIGHT**
  This property is used to indicate the light-filtering algorithm when PictureSmart™ is enabled. This property can be any of the following:

  - **PICTSMART_LIGHTCORRECTION_NONE**
    Indicates no light correction.
  - **PICTSMART_LIGHTCORRECTION_COOLFLORESCENT**
    Indicates a “blue” or “cool” florescent light.
  - **PICTSMART_LIGHTCORRECTION_WARMFLORESCENT**
    Indicates a “red” or “warm” florescent light.
  - **PICTSMART_LIGHTCORRECTION_OUTSIDE**
    Indicates an outside light.
  - **PICTSMART_LIGHTCORRECTION_TUNGSTEN**
    Indicates a tungsten light.

- **PROPERTY_MOTION_DETECTION_MODE**
  This property is used to enable/disable the motion detector. The motion detector sends notification events via the PortalNotification event (see **Notifications**). This property can be either TRUE or FALSE.

- **PROPERTY_MOTION_SENSITIVITY**
  This property is used to set the sensitivity of the motion detector. This property can range between 0..255.

- **PROPERTY_WHITELEVEL**
  This property is used adjust the whiteness of an image and is only available when auto-white leveling is turned off (see PROPERTY_AUTO_WHITELEVEL).

- **PROPERTY_AUTO_WHITELEVEL**
  This property controls auto-white leveling mode. When this property is set to ADJUSTMENT_AUTOMATIC, the auto-white level is engaged; otherwise auto-level mode is turned off. This property can be either ADJUSTMENT_MANUAL or ADJUSTMENT_AUTOMATIC.

- **PROPERTY_ANALOGGAIN**
  This property controls analog gainand is only available when auto-analog gain is turned off (see PROPERTY_AUTO_ANALOGGAIN). This property can range between 0..255.

- **PROPERTY_AUTO_ANALOGGAIN**
  This property controls auto-analog gain mode. When this property is set to ADJUSTMENT_AUTOMATIC, the auto-analog gain is engaged; otherwise auto-analog gain mode is turned off. This property can be either ADJUSTMENT_MANUAL or ADJUSTMENT_AUTOMATIC.

- **PROPERTY_LOWLIGHTBOOST**
  This property enables/disables low-light boosting. This property can be either TRUE or FALSE.

- **PROPERTY_COLORBOOST**
  This property enables/disables color boosting. This property can be either TRUE or FALSE.

- **PROPERTY_ANTIFLICKER**
  This property sets the anti-flicker mode. This property can be any of the following values:
  - **ANTIFLICKER_OFF**
    Turns off anti-flicker
  - **ANTIFLICKER_50Hz**
    Sets anti-flicker to 50Hz
  - **ANTIFLICKER_60Hz**
    Sets anti-flicker to 60Hz

- **PROPERTY_OPTIMIZATION_SPEED_QUALITY**
  This property sets the optimization setting. This property can be either OPTIMIZE_QUALITY, which indicates a higher quality image (slower), or OPTIMIZE_SPEED, which indicates a faster lower quality image.

- **PROPERTY_LED**
  This property sets current state of a camera's LED light. This property can be any of the following values:
  - **LED_OFF**
    Turns the LED light off.
  - **LED_ON**
    Turns the LED light on.
  - **LED_AUTO**
    The default LED light mode, which turns the LED light on when the camera is in use, and off otherwise.
  - **LED_MAX**

Appendix A

PROPERTY	VALUE	Logitech QuickCam Express	Logitech QuickCam Pro	Logitech QuickCam Home	Logitech QuickCam Web
PROPERTY_ORIENTATION	ORIENTATIONMODE_NORMAL	•	•	•	•
	ORIENTATIONMODE_MIRRORED	•	•	•	•
	ORIENTATIONMODE FLIPPED	•	•	•	•
	ORIENTATIONMODE_FLIPPED_AND_MIRRORED	•	•	•	•
PROPERTY_BRIGHTNESSMODE	ADJUSTMENT_MANUALADJUSTMENT_AUTOMATIC				
PROPERTY_BRIGHTNESS	0..255	•	•	•	•
PROPERTY_CONTRAST	0..255	•	•	•	•
PROPERTY_COLORMODE	ADJUSTMENT_MANUAL				
	ADJUSTMENT_AUTOMATIC				
PROPERTY_REDGAIN	0..255				
PROPERTY_BLUEGAIN	0..255				
PROPERTY_SATURATION	0..255	•	•	•	•
PROPERTY_EXPOSURE	0..255	•	•	•	•
PROPERTY_RESET	NO FLAGS	•	•	•	•
PROPERTY_ANTIBLOOM	**TRUE /** FALSE				
PROPERTY_LOWLIGHTFILTER	**TRUE /** FALSE				
PROPERTY_HUE	0..255	•	•	•	•
PROPERTY_PORT_TYPE	PORTTYPE_UNKNOWNPORTTYPE_LPT_NIBBLE				
	PORTTYPE_LPT_BIDIRECTIONAL				
	PORTTYPE_LPT_ECP				
	PORTTYPE LPT ECPDMA				
	PORTTYPE_USB	•	•	•	•
PROPERTY_PICTSMART_MODE	**TRUE /** FALSE		•		
PROPERTY_PICTSMART_LIGHT	PICTSMART_LIGHTCORRECTION_NONE				
	PICTSMART_LIGHTCORRECTION_COOLFLORESCENT				
	PICTSMART_LIGHTCORRECTION_WARMFLORESCENT				
	PICTSMART LIGHTCORRECTION OUTSIDE				
	PICTSMART_LIGHTCORRECTION_TUNGSTEN				
PROPERTY_MOTION_DETECTION_MODE	**TRUE /** FALSE	•	•	•	•
PROPERTY_MOTION_SENSITIVITY	0.255	•	•	•	•
PROPERTY_WHITELEVEL	0.255	•	•	•	•
PROPERTY_AUTO_WHITELEVEL	ADJUSTMENT_MANUAL	•	•	•	•
	ADJUSTMENT_AUTOMATIC	•	•	•	•
PROPERTY_ANALOGGAIN	0.255	•	•	•	•
PROPERTY_AUTO_ANALOGGAIN	ADJUSTMENT MANUAL	•	•	•	•
	ADJUSTMENT_AUTOMATIC	•	•	•	•
PROPERTY_LOWLIGHTBOOST	**TRUE /** FALSE	•	•	•	•
PROPERTY_COLORBOOST	**TRUE /** FALSE	•	•	•	•
PROPERTY_ANTIFLICKER	**TRUE /** FALSE	•	•	•	•
PROPERTY_OPTIMIZATION_SPEED_QUALITY	OPTIMIZE_QUALITYOPTIMIZE_SPEED			•	•
				•	•
PROPERTY_LED	LED OFFLED ONLED AUTOLED MAX				