



Programmer's Guide

# Quadro FX SDI

# Document Change History

Version	Date	Responsible	Reason for Change
01	January 24, 2008	TT, SM	Initial Release

# Table of Contents

<b>Programmer's Guide</b>	<b>1</b>
<b>Quadro FX SDI</b>	<b>1</b>
<b>Chapter 1. Getting Started</b>	<b>1</b>
<b>Chapter 2. Device Control APIs</b>	<b>2</b>
2.1. Windows XP	2
2.2. Linux	2
<b>Chapter 3. OpenGL Extensions</b>	<b>3</b>
<b>Chapter 4. Device Setup and Control</b>	<b>4</b>
4.1. Determining Video Capabilities	4
4.2. Opening the Video Device	5
4.3. Configuring the Video Device	6
<b>Chapter 5. Data Transfer</b>	<b>9</b>
5.1. Buffer Object Initialization	9
5.2. Pbuffer Initialization	12
5.3. Starting Video Transfers	18
5.4. Sending FBO Data	18
5.5. Sending Pbuffer Data	21
5.6. Stopping Video Transfers and Closing the Video Device	23
<b>Chapter 6. Video Compositing</b>	<b>25</b>
6.1. Alpha Compositing	25
6.2. Chroma-Keying	26
6.3. Luma-Keying	27
<b>Chapter 7. Changing the Video Device Configuration</b>	<b>28</b>
<b>Chapter 8. Device Feedback</b>	<b>29</b>
8.1. Determining the Number of Queued Buffers	29
8.1.1. Using the GLX/WGL_video_out Extension	30
8.1.2. Using the GL_present_video Extension	31
8.2. Detecting Duplicate Frames	33
8.2.1. Using the GLX/WGL_video_out Extension	33

8.2.2. Using the GL_present_video Extension .....	34
<b>Chapter 9. Advanced Topics .....</b>	<b>35</b>
9.1. Working with Two Video Channels .....	35
9.1.1. Dual-Link Operation .....	35
9.1.2. Two Independent Video Channels .....	35
9.2. Sending the Desktop to Video Output .....	36
9.3. Colorspace Conversion .....	37
9.3.1. Coefficients .....	38
9.3.2. Scale .....	38
9.3.3. Offset .....	39
9.4. Full-Scene Antialiasing .....	42
9.4.1. Pbuffer Multisampling .....	42
9.4.2. Multisampling with Buffer Objects .....	43
9.5. Calculating Video Memory Usage .....	45
9.6. Working with Greater than 8 Bits Per-Component .....	46
9.7. Data Integrity Check .....	46
9.8. Composite Sync Termination .....	48
9.9. Specifying the Internal Buffer Queue Length .....	49

## List of Tables

Table 5-1.	Pbuffer Size = Field .....	21
Table 5-2.	Pbuffer Size = Frame .....	22
Table 7-1.	Changeable and Unchangeable Configuration Parameters .....	28
Table 9-1.	SD ITU 601 Coefficients Table .....	38
Table 9-2.	HD ITU 709 Coefficients Table .....	38
Table 9-3.	Video Memory Required by an Application.....	45



# Chapter 1. Getting Started

Application programming of the NVIDIA® Quadro™ FX SDI is broken into two principle parts, device control and data transfer. Device control handles the hardware configuration as well as the starting and stopping of data transfers while data transfer is the sequence of operations that send graphics data to the video device for output.

The Quadro FX SDI displays graphics data that has been rendered by the graphics processing unit (GPU) into one or more OpenGL Frame Buffer Objects (FBO) or application controlled pbuffers using traditional OpenGL programming techniques. The resulting standard definition or high definition serial digital video output can be 8, 10, or 12-bit. 10-bit and 12-bit video output must originate from a 16-bit per-component floating-point render target while 8-bit data can originate from either an 8-bit per-component integer or a 16-bit per-component floating-point render target. Color data in an 8-bit FBO or pbuffer should be integer data in the range of 0 to 255, while data placed by an application into a 16-bit floating point pbuffer or FBO should be 16-bit float data in the range of 0 to 1.

The remainder of this document will outline the general procedures required to program all the Quadro FX SDI products. In cases where a feature may be only supported on a newer or more recent product, a note to this affect will appear in the text. For example, the Quadro FX 4600 and Quadro FX 5600 SDI represents the third generation graphics-to-SDI output board from NVIDIA. In addition to the traditional serial digital video output of GPU-rendered graphics, the Quadro FX 4600 and Quadro FX 5600 SDI supports the transfer of ancillary data within the video stream and basic 2D video compositing using an alpha, chroma or luma key.

**Note:** At this time, the API for ancillary data transfer is not complete. That functionality will be added in a future driver. At that time, an updated version of this document will be released.



# Chapter 2.

## Device Control APIs

---

### 2.1. Windows XP

On systems running the Microsoft Windows Operating System, hardware setup and control is handled by the NVIDIA Control Panel (NVCPL) API. Use of the NVCPL API requires the inclusion of the following include and library files. These files are packaged within the NVGVOSDK.

```
NvGvoApi.h  
NVApiError.h  
NvCpl.lib
```

Use of the NVCPL API to control the Quadro FX SDI device is described in *Chapter 4 Device Setup and Control*. For additional information on NVCPL, refer to the *NVCPL Programmers Guide*. This document can be downloaded from the NVIDIA Developer's Web site (<http://developers.nvidia.com>).

---

### 2.2. Linux

On Linux-based systems, hardware setup and control is enabled by the NV-CONTROL X extension. Use of the NV-CONTROL X extension requires the following include files. These files are packaged within the `nvidia-settings-1.0.tar.gz` archive that is included with the NVGVOSDK or display driver.

```
NVCtrlLib.h  
NVCtrl.h
```

Control of the Quadro FX SDI device with the NV-CONTROL X Extension is described in *Chapter 4 Device Setup and Control*. Additional information on the NV-CONTROL X Extension can be found in the NV-CONTROL-API.txt document included in the archive listed above.





## Chapter 3. OpenGL Extensions

Data transfer is enabled by extensions to OpenGL. The `GL_NV_present_video` extension provides a mechanism for the displaying of textures and renderbuffers on the Quadro FX SDI output. This extension is supported on both Windows XP and Linux systems. The `WGL_NV_video_out` extension sends pbuffers to the SDI device in the case of Windows XP while the `GLX_NV_video_out` extension provides the same capabilities on Linux systems. An application must utilize only one of the above extensions as the two extensions cannot be used together.

In addition to the above OpenGL extensions, other useful extensions include the following:

- ❑ `ARB_occlusion_query`
- ❑ `EXT_timer_query`
- ❑ `EXT_framebuffer_object`
- ❑ `ARB_pixel_format`
- ❑ `ARB_pbuffer`
- ❑ `NV_float_buffer`

**Note:** The `WGL_video_out` or `GLX_video_out` OpenGL extensions cannot be utilized with the `GL_present_video` OpenGL extension. An application must choose and utilize a single programming paradigm.

Additional information on these OpenGL extensions can be found in the extension specifications located at <http://developers.nvidia.com> or <http://opengl.org/>.



# Chapter 4.

## Device Setup and Control

Before graphics data can be transferred to the Quadro FX SDI for scan out as serial digital video, the video device must be properly configured for the desired video signal format, data format, timing, colorspace and genlock or frame lock synchronization. This hardware configuration is performed by the NVCPL API on Microsoft Windows-based systems and the NV-CONTROL X extension on Linux-based systems. The remainder of this section will describe the step by step process required to configure the video device. For additional information on the functions described, refer to the *NVCPL.DLL API Manual* or NV-CONTROL extension specification and included files.

---

### 4.1. Determining Video Capabilities

Prior to configuring a video device, query the device capabilities to confirm that the device class is defined as NVGVOCLASS\_SDI . The procedure for doing this on Windows using the NVCPL API is outlined in Code Listing 1.

#### Code Listing 1: Querying the Device Capabilities to Confirm Video Device Type.

```
const UINT nAdapterNumber = 1;
NVGVOCAPS gvoCaps = {0};
gvoCaps.cbSize = sizeof(gvoCaps);
if (NvGvoCaps(nAdapterNumber, 0, &gvoCaps) == NV_OK)
{
    if (gvoCaps.dwClass & NVGVOCLASS_SDI)
    {
        // Quadro FX 4000/4500/5500/4600/5600 SDI Available
    }

}
else
{
    printf("Failure: \'%s\'\n", NvGetLastErrorErrorMessage());
    exit(1);
}
```

On Linux, use the `XNVCTRLQueryAttribute` function to query `NV_CTRL_GVO_SUPPORTED` to determine if the X screen supports video output. This call will fail if video output is not supported on the current X screen, or if the video output is already in use by the desktop or another application.

### Code Listing 2: Using NV\_CONTROL-X Extension to Query Video Output Capabilities.

```

If (!XNVCTRLQueryAttribute(dpy, screen, 0,
                           NV_CTRL_GVO_SUPPORTED, &value)) {
    return FALSE;
} else {
    return TRUE;
}

```

## 4.2. Opening the Video Device

Once the availability of a video device has been confirmed, the next step is to open the video device. For video output, the device should be opened for exclusive write access as shown in Code Listing 3.

### Code Listing 3: Opening the Video Device.

```

NVGVOHANDLE hGVO = INVALID_NVGVOHANDLE;
if (NvGvoOpen(nAdapterNumber, NULL, NVGVOCLASS_SDI,
              NVGVO_O_WRITE_EXCLUSIVE, &hGVO) != NV_OK)
{
    // Handle error.
}

```

On Linux, the video device is part of the X screen, as such in the case that an X screen supports video output. The video device is automatically opened when the application opens the corresponding X Display. There is no need to issue a separate command to open the video device.

**Note:** This is a change since version 1.0 of the SDK. `NVGVOCLASS_P207` has been changed to `NVGVOCLASS_SDI`

## 4.3. Configuring the Video Device

After opening a video device, the device must be configured for the desired video output mode, format, timing, colorspace, genlock or frame lock characteristics and any other required video parameters. Code Listing 4 shows a simple example of device configuration for RGBA 4:4:4:4 input and 1080i YCrCbA 4:2:2:4 video output with composite tri-level sync.

**Note:** On the Quadro FX 4600 and Quadro FX 5600 SDI unlike earlier Quadro FX SDI products, because both the analog and digital SDI sync share a common BNC connector, the sync source (analog or SDI) must be specified, before the incoming signal format can be detected.

### Code Listing 4: Configuring a Video Device on Windows.

```
NVGVOCONFIG gvoConfig;
gvoConfig.cbSize = sizeof(gvoConfig);
gvoConfig.dwFields = NVGVOCONFIG_SIGNALFORMAT |
                    NVGVOCONFIG_DATAFORMAT |
                    NVGVOCONFIG_SYNCSOURCEENABLE |
                    NVGVOCONFIG_COMPOSITESYNCTYPE;
gvoConfig.signalFormat = NVGVOSIGNALFORMAT_1080I_5994_SMPTE274;
gvoConfig.dataFormat = NVGVODATAFORMAT_R8B8B8A8_TO_YCRCBA4224;
gvoConfig.syncEnable = TRUE;
gvoConfig.syncSource = NVGVOSYNCSOURCE_COMPSYNC;

NVGVOCHECK(NvGvoConfigSet(g_hGVO, &l_gvoConfig));

// Configure external sync parameters
DWORD l_dwWait;
NVGVOSTATUS l_gvoStatus;
l_gvoConfig.dwFields = 0; // reset fields

// Trigger redetection of sync format
l_nvResult = NvGvoSyncFormatDetect(g_hGVO, &l_dwWait);

// Wait for sync detection to complete
Sleep(l_dwWait);

// Get sync signal format
l_gvoStatus.cbSize = sizeof(NVGVOSTATUS);
l_nvResult = NvGvoStatus(g_hGVO, &l_gvoStatus);

// Verify that incoming sync signal is compatible outgoing signal
if (l_gvoStatus.syncFormat != l_gvoConfig.signalFormat) {

    // Handle Error
}
```

```
l_gvoConfig.syncEnable = l_gvoStatus.syncEnable;
l_gvoConfig.syncSource = l_gvoStatus.syncSource;

l_gvoConfig.compositeSyncType = NVGVOCOMPSYNCTYPE_AUTO;
l_gvoConfig.dwFields |= (NVGVOCONFIG_SYNCSOURCEENABLE |
                        NVGVOCONFIG_COMPOSITESYNCTYPE);

// Sync delay
NVGVOSYNCDELAY l_gvoSyncDelay;
memset(&l_gvoSyncDelay, 0, sizeof(l_gvoSyncDelay));
l_gvoSyncDelay.wHorizontalDelay = options->hDelay;
l_gvoSyncDelay.wVerticalDelay = options->vDelay;
l_gvoConfig.dwFields |= NVGVOCONFIG_SYNCDELAY;
l_gvoConfig.syncDelay = l_gvoSyncDelay;

NVGVOCHECK(NvGvoConfigSet(g_hGVO, &l_gvoConfig));
```

On Linux, the video device is configured using `XNVCTRLSetAttribute`. The following example configures the video device for RGBA 4:4:4:4 input and 1080i YCrCbA 4:2:2:4 video output with composite tri-level sync.

### Code Listing 5: Configuring a Video Device on Linux.

```
screen = DefaultScreen(dpy);

// Set video signal format
XNVCTRLSetAttribute(dpy, screen, 0,
                    NV_CTRL_GVO_OUTPUT_VIDEO_FORMAT,
                    NV_CTRL_GVO_VIDEO_FORMAT_1080I_59_94_SMPTE274);

// Set video data format
XNVCTRLSetAttribute(dpy, screen, 0,
                    NV_CTRL_GVO_DATA_FORMAT,
                    NV_CTRL_GVO_DATA_FORMAT_R8G8B8A8_TO_YCRCBA4224);

// Enable genlock
XNVCTRLSetAttribute(dpy, screen, 0,
                    NV_CTRL_GVO_SYNC_MODE,
                    NV_CTRL_GVO_SYNC_MODE_GENLOCK);

// Set sync type to composite.
XNVCTRLSetAttribute(dpy, screen, 0,
                    NV_CTRL_GVO_SYNC_SOURCE,
                    NV_CTRL_GVO_SYNC_SOURCE_COMPOSITE);

XFlush(dpy);

// Sleep to allow time for sync detection
sleep(2);

// Detect input sync.
XNVCTRLQueryAttribute(dpy, screen, 0,
                      NV_CTRL_GVO_COMPOSITE_SYNC_INPUT_DETECTED,
                      &val);

// If valid sync detected, query input video format.
If (val) {
    XNVCTRLQueryAttribute(dpy, screen
                          NV_CTRL_GVO_INPUT_VIDEO_FORMAT
                          &val);
}
```

# Chapter 5.

## Data Transfer

In programmable mode, the source for video output is data rendered into an 8-bit integer or 16-bit floating point frame buffer object (FBO) or pbuffer render target. For 10 or 12-bit video output, a 16-bit per-component floating-point render target must be utilized. For 8-bit output, either an 8-bit integer or 16-bit floating-point per-component render target may be used. An application may utilize a single render target or multiple render targets configured in a ring buffer in order to send data to a single video device. In order to send data to multiple video devices, multiple render targets are required.

This section describes the steps required to setup the render targets into which the application should render OpenGL. An application should use either frame buffer objects or pbuffers for data transfer and not combine the two techniques.

---

### 5.1. Buffer Object Initialization

Use of the `GL_NV_present_video` extension to send GPU rendered content to the SDI device requires the use of one or more buffer objects. These buffer objects can be texture objects or renderbuffers bound to one or more frame buffer objects (FBOs). Creation of an FBO with a texture or render buffer attachment is demonstrated in Code Listing 6.

#### Code Listing 6: Configuring a Frame Buffer Object.

```
GLuint fboId;
GLint textureObject;
GLuint renderbufferIds[2];
glGenRenderbuffersEXT(numRenderbuffers, renderbufferIds);

glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, renderbufferIds[0]);

glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT, GL_RGBA8,
                        width, height);

if (depth) {
    glBindRenderbufferEXT(GL_RENDERBUFFER_EXT,
                        renderbufferIds[1]);

    glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT,
                        GL_DEPTH_COMPONENT,
                        width, height);
}
```

```

glGenFramebuffersEXT(1, &fboId);

glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fboId);

if (!textureObject) {
    glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,
                                GL_COLOR_ATTACHMENT0_EXT,
                                GL_RENDERBUFFER_EXT,
                                renderbufferIds[0]);
} else {
    glBindTexture(GL_TEXTURE_RECTANGLE_NV, textureObject);
    glFramebufferTexture2D(GL_FRAMEBUFFER_EXT,
                           GL_COLOR_ATTACHMENT0_EXT,
                           GL_TEXTURE_RECTANGLE_NV,
                           textureObject, 0 );
}

if (depth) {
    glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,
                                GL_DEPTH_ATTACHMENT_EXT,
                                GL_RENDERBUFFER_EXT,
                                renderbufferIds[1]);
}

glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, 0);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

```

Creation of an FBO is identical on both Windows and Linux, and requires only a current OpenGL context. In the previous code listing example, when a texture object is specified, it is attached as `COLOR_ATTACHMENT0`, otherwise, a renderbuffer is used. For more information on FBO creation and usage, refer to the `GL_EXT_framebuffer_object` specification.

In order for an application to render into an FBO render target, the target must first be bound using the command in Code Listing 7.

### Code Listing 7: Configuring a Frame Buffer Object.

```
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fboId);
```



Either before or after buffer object creation, the SDI video device must be bound in OpenGL. The procedure for enumerating the available video devices and binding one of them is outlined in Code Listing 8.

### Code Listing 8: Binding a Video Device.

```
// Enumerate the available video devices and
// bind to the first one found
HVIDEOOUTPUTDEVICENV *videoDevices;

// Get list of available video devices.
int numDevices = wglEnumerateVideoDevicesNV(ghWinDC, NULL);

if (numDevices <= 0) {
    MessageBox(NULL, "wglEnumerateVideoDevicesNV() did not return
any devices.", "Error", MB_OK);
    exit(1);
}

videoDevices = (HVIDEOOUTPUTDEVICENV *)malloc(numDevices *
                                                sizeof(HVIDEOOUTPUTDEVICENV));

if (!videoDevices) {
    fprintf(stderr, "malloc failed. OOM?");
    exit(1);
}

if (numDevices != wglEnumerateVideoDevicesNV(ghWinDC,
                                              videoDevices)) {
    free(videoDevices);
    MessageBox(NULL, "Invonsistent results from
wglEnumerateVideoDevicesNV()", "Error", MB_OK);
    exit(1);
}

//Bind the first device found.
if (!wglBindVideoDeviceNV(ghWinDC, 1, videoDevices[0], NULL)) {
    free(videoDevices);
    MessageBox(NULL, "Failed to bind a videoDevice to slot 0.\n",
               "Error", MB_OK);
    exit(1);
}

// Free list of available video devices, don't need it anymore.
free(videoDevices);
```

## 5.2. Pbuffer Initialization

On Windows, the first step in the initialization of the pbuffer is to use the function `wglChoosePixelFormatARB()` to choose a compatible pixel format. For Quadro FX SDI video output from a pbuffer render target, the list of attributes specified as the second argument to this function must include `WGL_DRAW_TO_PBUFFER` as well as one of `WGL_BIND_TO_VIDEO_RGB_NV`, `WGL_BIND_TO_VIDEO_RGBA_NV`, or `WGL_BIND_TO_VIDEO_RGB_AND_DEPTH_NV`. Code Listing 9 and Code Listing 10 illustrate this procedure for both the 8 and 16-bit cases.

**Note:** In the 16-bit floating point per-component case, `WGL_FLOAT_COMPONENTS_NV` must also be specified in the attribute list.

### Code Listing 9: Choosing an 8-bit Pixel Format

```
int format = 0;
int nformats = 0;
int attribList = {
    WGL_RED_BITS_ARB, 8,
    WGL_GREEN_BITS_ARB, 8,
    WGL_BLUE_BITS_ARB, 8,
    WGL_ALPHA_BITS_ARB, 8,
    WGL_STENCIL_BITS_ARB, 8,
    WGL_DEPTH_BITS_ARB, 24,
    WGL_DRAW_TO_PBUFFER_ARB, true,
    WGL_BIND_TO_VIDEO_RGBA_NV, true,
    0 };

wglChoosePixelFormatARB(hDC, attribList, 1, &format, &nformats);
```

### Code Listing 10: Choosing a 16-bit Floating Point Pixel Format

```
int format = 0;
int nformats = 0;
int attribList = {
    WGL_RED_BITS_ARB, 16,
    WGL_GREEN_BITS_ARB, 16,
    WGL_BLUE_BITS_ARB, 16,
    WGL_ALPHA_BITS_ARB, 16,
    WGL_STENCIL_BITS_ARB, 8,
    WGL_DEPTH_BITS_ARB, 24,
    WGL_DRAW_TO_PBUFFER_ARB, true,
    WGL_BIND_TO_VIDEO_RGBA_NV, true,
    WGL_FLOAT_COMPONENTS_NV, true,
    0 }

wglChoosePixelFormat(hDC, attribList, 1, &format, &nformats);
```

Once an available pixel format that meets the requirements has been specified, a pbuffer can be created using the `wglCreatePbufferARB()` function. For video output, the attribute list specified as the final argument must contain one of `WGL_BIND_TO_VIDEO_RGB_NV`, `WGL_BIND_TO_VIDEO_RGBA_NV`, or `WGL_BIND_TO_VIDEO_RGB_AND_DEPTH_NV`. Code Listing 11 demonstrates pbuffer creation.

### Code Listing 11: Creating a Pbuffer.

```
attribList[0] = WGL_BIND_TO_VIDEO_RGBA_NV;
attribList[1] = true;
attribList[2] = 0;

hPbuf = wglCreatePbufferARB(hDC, format, width, height,
                           attribList);
```

Once one or more pbuffers has been created using the above procedure, each of these pbuffers must be bound to the video device so that subsequent OpenGL rendering is sent to the video output device. Prior to binding however, it is necessary to identify the video devices available using the `wglGetVideoDeviceNV()` function from the `WGL_NV_video_out` extension..

### Code Listing 12: Get Video Devices Available on the System.

```
HPVIDEODEV hpDevList;

if ((wglGetVideoDeviceNV(hDC, 1, &hpDevList) != GL_NOERROR) {

    // Handle error.
}
```

Once a video device has been identified, bind the application pbuffers using the `wglBindVideoImageNV()` command. The final argument to this function must be one of `WGL_VIDEO_OUT_COLOR_NV`, `WGL_VIDEO_OUT_ALPHA_NV`, `WGL_VIDEO_OUT_COLOR_AND_ALPHA_NV` or `WGL_VIDEO_OUT_COLOR_AND_DEPTH_NV`. This argument specifies the data within the pbuffer that will ultimately get transferred to the video out device. In Code Listing 13, the color and alpha portions of the pbuffer specified by `hPbuf` are bound to the video device.

### Code Listing 13: Binding a Pbuffer to a Video Device.

```
if ((wglBindVideoImageNV(hPdevList, hPbuf,
                        WGL_VIDEO_OUT_COLOR_AND_ALPHA) != GL_NOERROR) {

    // Handle error.
}
```

On Linux, a pbuffer is created using a combination of `glXChooseFBConfig()` to choose compatible frame buffer configurations and `glXCreatePbuffer()` to create a pbuffer of the width and height required. This process for both an 8-bit integer and a 16-bit floating point pbuffer is outlined in Code Listing 14 and Code Listing 15.

**Note:** Once `glXChooseFBConfig` returns the list of compatible frame buffer configurations, an application must traverse the list to find a configuration of the desired color depth.

### Code Listing 14: Creating an 8-bit Pbuffer on Linux.

```
GLXFBConfig *configs, config;
int nelements;
int config_list[] = { GLX_DRAWABLE_TYPE, GLX_PBUFFER_BIT,
                     GLX_ALPHA_SIZE, 8,
                     GLX_DOUBLE_BUFFER, GL_TRUE,
                     GLX_RENDER_TYPE, GLX_RGBA_BIT,
                     None };
int pbuffer_list[8];

configs = glXChooseFBConfig(dpy, 0, config_list, &nelements);

// Find a config with the right number of color bits.
for (i = 0; i < nelements; i++) {

    int attr;
    if (glXGetFBConfigAttrib(dpy, configs[i],
                           GLX_RED_SIZE, &attr)) {
        // Handle error
    }

    if (attr != 8)
        continue;

    if (glXGetFBConfigAttrib(dpy, configs[i],
                           GLX_GREEN_SIZE, &attr)) {
        // Handle error
    }

    if (attr != 8)
        continue;

    if (glXGetFBConfigAttrib(dpy, configs[i],
                           GLX_BLUE_SIZE, &attr)) {
        // Handle error
    }

    if (attr != 8)
        continue;

    if (glXGetFBConfigAttrib(dpy, configs[i],
                           GLX_ALPHA_SIZE, &attr)) {
        // Handle error
    }
}
```

```

        if (attr != 8)
            continue;

        break;
    }

    if (i == nelements) {
        printf("No 8-bit FBConfigs found\n");
        return -1;
    }

    // Config found
    config = configs[i];

    // Don't need the config list anymore so free it.
    XFree(configs);
    configs = NULL;

    pbuffer_list[0] = GLX_PBUFFER_WIDTH;
    pbuffer_list[1] = gWidth;
    pbuffer_list[2] = GLX_PBUFFER_HEIGHT;
    pbuffer_list[3] = gHeight;
    pbuffer_list[4] = None;

    pbuffer = glXCreatePbuffer(dpy, config, pbuffer_list);

    // Create rendering context for GLX_RGBA_TYPE pbuffer.
    context = glXCreateNewContext(dpy, config,
                                  GLX_RGBA_TYPE, 0, True);

```

### Code Listing 15: Creating a 16-bit Floating Point Pbuffer on Linux.

```

GLXFBConfig *configs, config;
int nelements;
int config_list[] = { GLX_DRAWABLE_BIT, GLX_PBUFFER_BIT,
                      GLX_DOUBLEBUFFER, GL_TRUE,
                      GLX_RENDER_TYPE, GLX_RGBA_FLOAT_BIT_ARB,
                      GLX_RED_SIZE, 16,
                      GLX_GREEN_SIZE, 16,
                      GLX_BLUE_SIZE, 16,
                      None };

int pbuffer_list[8];

configs = glXChooseFBConfig(dpy, 0, config_list, &nelements);

// Find a config with the right number of color bits.
for (i = 0; i < nelements; i++) {

    int attr;
    if (glXGetFBConfigAttrib(dpy, configs[i],
                            GLX_RED_SIZE, &attr)) {

        // Handle error
    }
}

```

```

    }

    if (attr != 16)
        continue;

    if (glXGetFBConfigAttrib(dpy, configs[i],
                           GLX_GREEN_SIZE, &attr)) {
        // Handle error
    }

    if (attr != 16)
        continue;

    if (glXGetFBConfigAttrib(dpy, configs[i],
                           GLX_BLUE_SIZE, &attr)) {
        // Handle error
    }

    if (attr != 16)
        continue;

    if (glXGetFBConfigAttrib(dpy, configs[i],
                           GLX_ALPHA_SIZE, &attr)) {
        // Handle error
    }

    if (attr != 16)
        continue;

    break;
}

if (i == nelements) {
    printf("No 16-bit FBConfigs found\n");
    return -1;
}

// Config found
config = configs[i];

// Don't need the config list anymore so free it.
XFree(configs);
configs = NULL;

pbuffer_list[0] = GLX_PBUFFER_WIDTH;
pbuffer_list[1] = gWidth;
pbuffer_list[2] = GLX_PBUFFER_HEIGHT;
pbuffer_list[3] = gHeight;
pbuffer_list[4] = None;

// Create pbuffer
pbuffer = glXCreatePbuffer(dpy, config, pbuffer_list);

// Create rendering context for GLX_RGBA_FLOAT_TYPE_ARB pbuffer.

```

```
context = glXCreateNewContext(dpy, config,
                             GLX_RGBA_FLOAT_TYPE_ARB, 0, True);
```

**Note:** Checking the proper color depth of the chosen FBConfig is required as glXChooseFBConfig in recent Linux drivers returns deeper FBConfigs at the beginning of the resulting list. As a result 32-bit FBConfigs appear earlier in the list than FP16 or 8-bit integer configs. The change was made to bring the behavior of glXChooseFBConfig inline with the specification.

After creating one or more pbuffers using the procedure described in Code Listing 15, each of these pbuffers must be bound to the video device so that subsequent OpenGL rendering into that pbuffer is sent to the video output device. Prior to binding the pbuffer however, it is necessary to identify the video devices available using the glXGetVideoDeviceNV() function from the GLX\_NV\_video\_out extension.

### Code Listing 16: Getting Video Devices Available on a Linux System.

```
GLXVideoDeviceNV video_device;

if (glXGetVideoDeviceNV(dpy, 0, 1, &video_device) {
    // Handle error.
}
```

Once a video device has been identified, bind the application pbuffers using the glXBindVideoImageNV() command. The final argument to this function must be one of GLX\_VIDEO\_OUT\_COLOR\_NV, GLX\_VIDEO\_OUT\_ALPHA\_NV, GLX\_VIDEO\_OUT\_COLOR\_AND\_ALPHA\_NV or GLX\_VIDEO\_OUT\_COLOR\_AND\_DEPTH\_NV. This argument specifies the data within the pbuffer that will ultimately get transferred to the video out device. In Code Listing 17, the color and alpha portions of the pbuffer specified by hPbuf are bound to the video device.

### Code Listing 17: Binding a Pbuffer to a Linux Video Device.

```
if (glXBindVideoImageNV(dpy, video_device,
                        pbuffer, GLX_VIDEO_OUT_COLOR_NV)) {
    // Handle error.
}
```

Once the pbuffers are bound to the video device, video transfers can be started as described in *Starting Video Transfers*. The procedure for sending frames/fields of data to the device is described in the section entitled *Sending FBO Data and Sending Pbuffer Data*.



## 5.3. Starting Video Transfers

Once a video device has been configured, and the OpenGL pBuffer(s) required for data transfer have been allocated and bound to the device, the next step is to commence video transfers. This step is only required on Windows. The code to do this is listed in Code Listing 18.

### Code Listing 18: Starting VideoTransfers

```
If ((hGVO) && !(NvGvoIsRunning(hGVO)))
{
    if (NvGvoStart(hGVO) != NV_OK)
    {
    }
}
```

## 5.4. Sending FBO Data

Once GPU rendering is complete, the contents of the render buffer or texture objects is queued to the SDI video device with either the `glPresentFrameKeyedNV()` or `glPresentFrameDualFillNV()` functions.

```
void glPresentFrameKeyedNV(uint video_slot,
                           uint64EXT minPresentTime,
                           uint beginPresentTimeId,
                           uint presentDurationId,
                           enum type,
                           enum target0, uint fill0, uint key0,
                           enum target1, uint fill1, uint key1);
```

`glPresentFrameKeyedNV` should be utilized to display single- or dual-link fill or fill and key data. The `video_slot` parameter specifies the video output slot in the current rendering context on which this frame should be presented. The value of `minPresentTime` should be set to either the earliest time in nanoseconds that the frame should become visible on the SDI output, or the special value of 0 which indicates that the frame should be presented at the next vertical retrace.

Frame presentation is always queued until the vertical blanking period of the SDI device. At that time, the SDI device will display the:

- ❑ Last presented frame if there are no additional frames queued to present.
- ❑ Next frame in the queue with the minimum presentation time of 0.
- ❑ The last frame in the queue that has a minimum presentation time past the current time.

Queued frames are always consumed in the order in which they were sent. Any consumed frames not displayed are discarded.

The values of `beginPresentTimeId` and `presentDurationId` represent valid query objects. These should be utilized to query the time at which the frame is displayed on the SDI device and the number of vertical retrace intervals that the frame was presented. Use of these query objects will be discussed in more detail in Section 8.1.2. These values should be set to 0 if they are unused.

The parameter `type` indicates the type of data to be displayed. Valid values for `type` are `GL_FIELDS_NV` and `GL_FRAME_NV`. When `GL_FIELDS_NV` is specified, both fields must be specified by `target0`, `fill0`, `key0`, `target1`, `fill1` and `key1` in order to complete the frame. In the case of `GL_FRAME_NV`, only `target0`, `fill0` and `key0` should be utilized to specify the frame data.

The parameters `target0` and `target1` indicate the data object type and can be one of `GL_TEXTURE_2D`, `GL_TEXTURE_RECTANGLE`, `GL_RENDERBUFFER_EXT` or `GL_NONE`. The values of `fill0` and `fill1` name the objects of the corresponding type from which the color data will be read while `key0` and `key1` name the objects from which the key channel data will be read. In the case that there is no key channel data to display, the values of `key0` and `key1` should be set to `GL_NONE`. In this case, the key data is taken from the alpha channel of the specified fill object. In the case that the type specified is `GL_FRAME_NV`, `target1`, `fill1` and `key1` should be specified as `GL_NONE`.

```
void glPresentFrameDualFillNV(uint video_slot,
                             uint64EXT minPresentTime,
                             uint beginPresentTimeId,
                             uint presentDurationId,
                             enum type,
                             enum target0, uint fill0,
                             enum target1, uint fill1,
                             enum target2, uint fill2,
                             enum target3, uint fill3);
```

`glPresentFrameDualFillNV( )` should be utilized to display two channels of single-link fill data. This operating mode is useful for presenting two completely different fill channels or two fill channels in stereo, where one eye is presented on one channel while the other eye is presented on the other SDI channel.

`glPresentFrameDualFillNV( )` operates similarly to `PresentFrameKeyedNV( )` described above. The parameters `target0`, `fill0`, `target1` and `fill1` specify the data for the first SDI channel while `target2`, `fill2`, `target3` and `fill3` specify the data for the second SDI channel. In the case that `type` is `GL_FRAME_NV`, only `target0`, `fill0`, `target2` and `fill2` need to be specified with other parameters set to `GL_NONE`. In the case of `GL_FIELDS_NV`, all parameters must be utilized to specify the fields for each SDI channel.

The following code examples demonstrate the use of these functions. For more information regarding these functions, please refer to the `GL_NV_present_video` OpenGL extension specification.

In Code Listing 19, the contents of a texture object is interpreted as a complete frame with the `GL_FRAME_NV` enum and displayed as both the video fill and key channels.

### Code Listing 19: Sending a Texture Object to the Video Device

```
// Unbind frame buffer object
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

// Send texture object to SDI device
glPresentFrameKeyedNV(1, 0,
                      presentTimeID, presentDurationID,
                      GL_FRAME_NV,
                      GL_TEXTURE_RECTANGLE_NV, gTO, 0,
                      GL_NONE, 0, 0);
```

This code example, specifies that the texture object should be displayed on video slot 1 at the next vertical retrace. The query objects `presentTimeID` and `presentDurationID` return the time at which the frame is displayed on the SDI device and the number of vertical retrace intervals during which the frame is displayed. These timer query objects can then be utilized to determine if frame has been dropped or duplicated. More information about synchronization and catching irregularities can be found in Section 8.1.2.

Code Listing 20 specifies the fill for two video channels for the case in which the Quadro FX SDI is configured to display two fill channels rather than a fill and a key channel. In this example, two render buffers containing complete frames are displayed on video slot 1 at the next vertical retrace of the SDI device.

#### Code Listing 20: Sending Two Video Fill Channels

```
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo1Id);
drawPattern1();
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo2Id);
drawPattern2();
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

glPresentFrameDualFillNV(1, 0, 0, 0, GL_FRAME_NV,
                        GL_RENDERBUFFER_EXT, renderbuffer1Id,
                        GL_NONE, 0,
                        GL_RENDERBUFFER_EXT, renderbuffer2Id,
                        GL_NONE, 0);
```

## 5.5. Sending Pbuffer Data

On Windows, an application sends graphics data to the video device by rendering frames or fields into a pbuffer and then calling `wglSendPbufferToVideoNV()` to indicate that OpenGL rendering is complete and that the pbuffer is ready to be scanned out by the video device. The call to this function should typically be placed at the end of the application draw loop.

```
BOOL wglSendPbufferToVideoNV (HPBUFFERARB hPbuffer,
                              int iBufferType,
                              unsigned long *pulCounterPbuffer,
                              BOOL bBlock);
```

The parameter `hPbuffer` is the handle of the pbuffer. Both frames and fields can be sent to the video device as indicated by the value of `iBufferType`, and the relative size of the pbuffer as outlined in Tables 5-1 and 5-2.

Table 5-1. Pbuffer Size = Field

<b>iBufferType</b>	<b>Pbuffer Size = Field</b>
WGL_VIDEO_OUT_FIELD_1	Field 0
WGL_VIDEO_OUT_FIELD_2	Field 1
WGL_VIDEO_OUT_FRAME	Half height frame
WGL_VIDEO_OUT_STACKED_FIELDS_1_2	Half height frame
WGL_VIDEO_OUT_STACKED_FIELDS_2_1	Half height frame

Table 5-2. Pbuffer Size = Frame

iBufferType	Pbuffer Size = Frame
WGL_VIDEO_OUT_FIELD_1	Even lines to field 0
WGL_VIDEO_OUT_FIELD_2	Odd lines to field 1
WGL_VIDEO_OUT_FRAME	Full frame
WGL_VIDEO_OUT_STACKED_FIELDS_1_2	Upper half to field 2 Bottom half to field 1
WGL_VIDEO_OUT_STACKED_FIELDS_2_1	Upper half to field 1 Bottom half to field 2

The parameter `pulCounterPbuffer` returns the count of the pbuffer sent. This value should always increment and can be used to calculate if frames or fields have been dropped or duplicated as described in Section 8.1.1. The value of `bBlock` indicates if the function should queue the pbuffer and return immediately or wait until the pbuffer is actually scanned out by the SDI device. The following example sends a frame of graphics data to the video device.

### Code Listing 21: Sending a Frame of Data to the Video Device

```
if ((wglSendPbufferToVideoNV(hpBuf, WGL_VIDEO_OUT_FRAME,
    &gBufCount, TRUE) != GL_NOERROR) {
    // Handle error.
}
```

In this example, the fourth argument to `wglSendPbufferToVideoNV()` is set to **TRUE** to indicate that this function call should block until video scan out of this frame is complete. This aids in the synchronization of graphics and video in applications that utilize only a single pbuffer.

**Note:** Color data in an 8-bit pbuffer should be integer data in the range of 1 to 256 while data placed by an application into a 16-bit floating point pbuffer should be 16-bit float data in the range of 0 to 1.

On Linux, the procedure of sending a pbuffer containing a field or frame to the video output is identical to that for Windows except for the function used. On Linux, the function `glXSendPbufferToVideoNV()` is utilized as illustrated in the following example.

```
int glXSendPbufferToVideoNV(Display *dpy, GLXPbuffer pbuf,
    int iBufferType,
    unsigned long *pulCounterPbuffer,
    GLboolean bBlock);
```

### Code Listing 22: Sending a Frame of Data to the Linux video device.

```
glXSendPbufferToVideoNV(dpy, pbuffer,
    GLX_VIDEO_OUT_FRAME_NV,
    &gBufCount, GL_FALSE);
```

## 5.6. Stopping Video Transfers and Closing the Video Device

On Windows, once an application has completed all video transfers and no longer needs access to a video device, an application should stop sending and release any OpenGL resources prior to stopping and closing the device with the NvCpl API. If the WGL\_video\_out OpenGL extension is in use, the pBuffer and video device should be released as shown in Code Listing 23. The following code examples demonstrate the order in which functions should be called in order to safely release and close a video device.

### Code Listing 23: Releasing Bound OpenGL Resources.

```
wglReleaseVideoImageNV(gpBuffer.getHandle(),
    WGL_VIDEO_OUT_COLOR_AND_ALPHA_NV);

wglReleaseVideoDeviceNV(ghpDevList[0]);
```

In the case of the GL\_present\_video extension, no formal releasing of OpenGL resources is required. However, in both cases, once the OpenGL resources are released, then the video device can be stopped and closed as shown in Code Listing 24.

### Code Listing 24: Closing the Video Device on Windows.

```
if (hGVO)
{
    if (NvGvoIsRunning(hGVO))
    {
        NvGvoStop(hGVO);
    }
    if (NvGVOClose(hGVO) != NV_OK) {
    }
}
```

On Linux there are no requirements within the CONTROL-X extension to stop video transfers and close the device. However, one must unbind any associated pbuffers and release the video device using the appropriate GLX\_NV\_video\_out functions, prior to deleting the pbuffers. This is outlined Code Listing 25.

### Code Listing 25: Releasing Bound OpenGL Resources on Linux.

```
glXReleaseVideoImageNV(dpy, pbuffer);  
  
glXReleaseVideoDeviceNV(dpy, 0, video_device);  
  
glXDestroyPbuffer(dpy, pbuffer);
```



# Chapter 6.

## Video Compositing

The Quadro FX 4600 and Quadro FX 5600 support programmable 2D compositing. This operating mode combines the image data from the incoming video stream with the GPU-rendered image based upon the values in a third image known as a matte or key channel. The Quadro FX 4600 and Quadro FX 5600 SDI supports the following compositing methods.

**Note:** 2D compositing is only supported when the signal format of the incoming video matches the outgoing signal format and the outgoing SDI video signal is genlocked to the input signal. Scaling or retiming the video input to match the video output signal format is not supported.

### 6.1. Alpha Compositing

In the case of alpha compositing, the application provides the key / matte channel per field / frame in the alpha channel of the OpenGL graphics stream. When the application enables alpha compositing via the API, the Quadro FX 4600 and Quadro FX 5600 SDI simply executes the function below to compute the final pixel color.

***output color = input video color \* (1 - alpha) + input graphics color \* (alpha)***

When the value of the key or matte is strictly 1 or 0, the compositing is complete replacement. When the key value is between 0 and 1, the two images (video and graphics) would be blended. The above formula is executed for each and every pixel in the SDI output stream.

In configuration of the SDI device, alpha compositing is enabled as follows:

```
l_gvoConfig.dwFields = 0;
l_gvoConfig.dwFields = NVGVOCONFIG_COMPOSITE;
l_gvoConfig.bEnableComposite = TRUE;
l_gvoConfig.dwFields = NVGVOCONFIG_ALPHAKEYCOMPOSITE;
l_gvoConfig.bEnableAlphaKeyComposite |= TRUE;
NVGVOCHECK(NvGvoConfigSet(g_hGVO, &l_gvoConfig));
```

OpenGL must be configured to provide an alpha channel to the SDI device. In the case of sending pbuffer data with WGL/GLX\_send\_video this is done by specifying VIDEO\_OUT\_COLOR\_AND\_ALPHA to the bind command as follows:

```
// Bind pbuffer to video device.
if (!wglBindVideoImageNV(ghpDevList[0], gPBuffer.getHandle(),
    WGL_VIDEO_OUT_COLOR_AND_ALPHA_NV)) {
    // Handle error.
}
```

In the case of the GL\_present\_video extension, the format of the buffer object or texture must contain an alpha channel. Typical formats to create such a buffer object or texture would be either GL\_RGBA8 or GL\_RGBA16F\_ARB.

## 6.2. Chroma-Keying

To perform chroma keying, the application can specify up to two (Cr,Cb) pairs that represent the starting and ending chroma values for replacement within the video stream. Chroma keying is enabled with the ranges specified using the control API as follows.

```
// Cr composite ranges
l_gvoConfig.dwFields = 0;          // reset fields
l_gvoConfig.dwFields = NVGVOCONFIG_COMPOSITE |
    NVGVOCONFIG_COMPOSITE_CR;
l_gvoConfig.bEnableComposite = TRUE;
l_gvoConfig.compRange.bEnabled = TRUE;

l_gvoConfig.compRange.dwRange = 0;
l_gvoConfig.compRange.dwMin = options->crCompRange[0];
l_gvoConfig.compRange.dwMax = options->crCompRange[1];
NVGVOCHECK(NvGvoConfigSet(g_hGVO, &l_gvoConfig));

l_gvoConfig.compRange.dwRange = 1;
l_gvoConfig.compRange.dwMin = options->crCompRange[2];
l_gvoConfig.compRange.dwMax = options->crCompRange[3];
NVGVOCHECK(NvGvoConfigSet(g_hGVO, &l_gvoConfig));

// Cb composite ranges
l_gvoConfig.dwFields = 0;          // reset fields
l_gvoConfig.dwFields = NVGVOCONFIG_COMPOSITE |
    NVGVOCONFIG_COMPOSITE_CB;
l_gvoConfig.bEnableComposite = TRUE;
l_gvoConfig.compRange.bEnabled = TRUE;

l_gvoConfig.compRange.dwRange = 0;
l_gvoConfig.compRange.dwMin = options->cbCompRange[0];
l_gvoConfig.compRange.dwMax = options->cbCompRange[1];
NVGVOCHECK(NvGvoConfigSet(g_hGVO, &l_gvoConfig));

l_gvoConfig.compRange.dwRange = 1;
l_gvoConfig.compRange.dwMin = options->cbCompRange[2];
```

```
l_gvoConfig.compRange.dwMax = options->cbCompRange[3];
NVGVOCHECK(NvGvoConfigSet(g_hGVO, &l_gvoConfig));
```

When the application enables chroma keying, the Quadro FX 4600 and Quadro FX 5600 examines the chroma (Cr and Cb) for each incoming video pixel. If the (Cr, Cb) value for that pixel is within one of the specified ranges, the color of that pixel is replaced with the pixel color from the GPU.

---

## 6.3. Luma-Keying

For luma keying, the application specifies up to two pairs of luma (Y) values via the API. These values represent the starting and ending luma values for replacement. Luma keying is enabled with the range specified as follows.

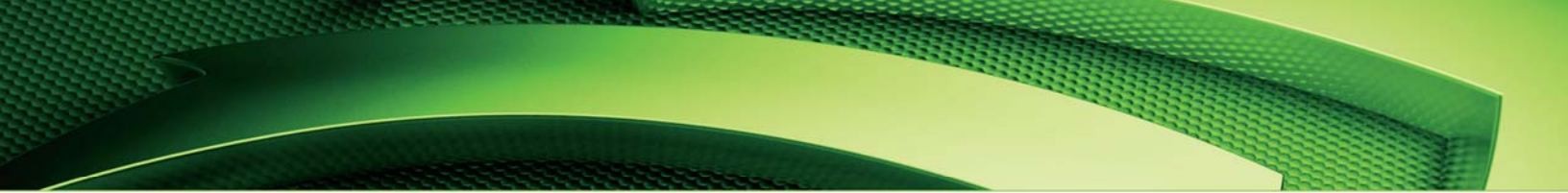
```
// Y composite ranges
l_gvoConfig.dwFields = 0;          // reset fields

l_gvoConfig.dwFields = NVGVOCONFIG_COMPOSITE |
                      NVGVOCONFIG_COMPOSITE_Y;
l_gvoConfig.bEnableComposite = TRUE;
l_gvoConfig.compRange.bEnabled = TRUE;

l_gvoConfig.compRange.dwRange = 0;
l_gvoConfig.compRange.dwMin = options->yCompRange[0];
l_gvoConfig.compRange.dwMax = options->yCompRange[1];
NVGVOCHECK(NvGvoConfigSet(g_hGVO, &l_gvoConfig));

l_gvoConfig.compRange.dwRange = 1;
l_gvoConfig.compRange.dwMin = options->yCompRange[2];
l_gvoConfig.compRange.dwMax = options->yCompRange[3];
NVGVOCHECK(NvGvoConfigSet(g_hGVO, &l_gvoConfig));
```

When luma compositing is enabled, the Quadro FX 4600 and the Quadro FX 5600 compares the luma (Y) for each incoming video pixel to the specified ranges. If the value for that pixel is within a specified range, the color of that pixel is replaced with the pixel color from the GPU.



## Chapter 7. Changing the Video Device Configuration

Changes to the SDI video device configuration can be made using `NvGvoConfigSet()` and `XNVCTRLSetAttribute()` as outlined in the section entitled *Configuring the Video Device*. Some configuration parameters may be changed during video transfers while for other changes, video output must first be stopped with OpenGL resources released before the new configuration parameters can take affect. Table 7-1 shows those parameters that maybe changed during video transfers and those that cannot be changed during video transfers.

Table 7-1. Changeable and Unchangeable Configuration Parameters

Changeable	Unchangeable
Color Space Conversion	Signal Format
Gamma Correction	Data Format
Synchronization	Flip Queue Length
Compositing	



## Chapter 8. Device Feedback

The `WGL_NV_video_out/GLX_NV_video_out` as well as the `GL_NV_present_video` extensions provide device feedback. This functionality provides the following information to the calling application.

- ❑ Number of buffers queued for SDI scanout.
- ❑ Vertical retrace interval count.
- ❑ Time at which a frame was scanned out.
- ❑ Number of vertical retrace intervals during which the same frame was scanned out.

This information can subsequently be used by an application to determine the video transfer latency within the system and if frames on the SDI output have been dropped or duplicated. Applications can then use this information to make adjustments in the data sent to the SDI device.

---

### 8.1. Determining the Number of Queued Buffers

The driver software for the Quadro FX SDI maintains a collection of internal buffers. These internal buffers permit an application to queue a frame and continue in order to begin processing future frames. The default number of buffers is five. To change this default number of internal buffers, see the advanced topic *Specifying Internal Buffer Queue Length*, later in this document.

### 8.1.1. Using the GLX/WGL\_video\_out Extension

Applications that utilize the `wglSendPbufferToVideoNV()` or `glXSendPbufferToVideoNV()` functions within the `WGL_video_out` or `GLX_video_out` extensions should set the `bBlock` argument to `GL_FALSE` in order to queue buffers, otherwise if blocking is specified, the number queued buffers will always be 1. An application should then call `wglGetVideoInfoNV()` or `glXGetVideoInfoNV()` in order to retrieve the number of vertical blanks since the start of video transfers along with the count of the current buffer being scanned out.

```

BOOL wglGetVideoInfoNV (HPVIDEODEV hpVideoDevice,
                        unsigned long *pulCounterOutputVideo,
                        unsigned long *pulCounterOutputPbuffer);

int glXGetVideoInfoNV(Display *dpy, int screen,
                     GLXVideoDeviceNV VideoDevice,
                     unsigned long *pulCounterOutputVideo,
                     unsigned long *pulCounterOutputPbuffer);

```

To determine the current number of full buffers queued, an application should subtract the value of `pulCounterOutputVideo` from the value of `pulCounterPbuffer` returned from `wglSendPbufferToVideoNV()` or `glXSendPbufferToVideoNV()` as outlined in Code Listing 26.

#### Code Listing 26: Determining the number of queued buffers.

```

unsigned long l_ulNumVertBlanks;
unsigned long l_ulNumBufs;
int l_uiNumBufsQueued = 0;

wglGetVideoInfoNV(ghpDevList[0], &l_ulNumBufs,
                  &l_ulNumVertBlanks);
.
.
.
l_bRes = wglSendPbufferToVideoNV(hpBuf, l_iBufType, &gBufCount,
                                FALSE);
l_uiNumBufsQueued = gBufCount - l_ulNumBufs;

```

## 8.1.2. Using the GL\_present\_video Extension

In the case of applications that utilize `glPresentFrameKeyedNV()` or `glPresentFrameDualFillNV()` provided by the `GL_NV_present_video` extension, the number of queued buffers can be determined by subtracting the time at which the buffer was sent from the time at which the buffer was presented or scanned out and then divide this value by the presentation interval.

The time at which a buffer is sent can be determined utilizing one of the query functions below by specifying `pname` as `GL_CURRENT_TIME_NV`.

```
void GetVideoivNV(uint video_slot, enum pname,
                  int *params);
```

```
void GetVideoiivNV(uint video_slot, enum pname,
                   uint *params);
```

```
void GetVideoi64vNV(uint video_slot, enum pname,
                    int64EXT *params);
```

```
void GetVideoi64vNV(uint video_slot, enum pname,
                    uint64EXT *params);
```

The current time on the Quadro FX SDI output device in nanoseconds is then returned in the value of `params`.

The present or scanout time for a particular buffer is determined by querying, using one of the functions above with the query target `GL_PRESENT_TIME_NV`, the value of the query object specified as `beginPresentTimeID` in the call to `glPresentFrameKeyedNV()` or `glPresentFrameDualFillNV()`. The value returned is then the time in nanoseconds that the frame first started scanning out. The results of the query will not be available until frame starts scanning out. For that reason, it is best to manage query objects as a circular buffer as outlined in Code Listing 27.

The presentation interval can be determined by subtracting the current present time from the last present time. The resulting value is the time between the scanout of two subsequent frames. This time should match the expected time for the chosen video signal format. For example, the presentation interval for 1080i5994 should be roughly 33 msec while for 720p60 the presentation interval should be 16 msec, and the presentation for 1080p24 should be approximately 40 msec.



## Code Listing 27: Determining Number of Buffers Queued.

```

static int cur_query = 0;
static bool queryTime = GL_FALSE;
GLuint64EXT presentTime;
static GLuint64EXT lastPresentTime = 0;
GLuint durationTime;
static GLuint64EXT sendTime[NUM_QUERIES];
GLuint presentTimeID = gPresentID[cur_query];
GLuint presentDurationID = gDurationID[cur_query];

cur_query++;

// Query video present time and duration.  Only do this once
// we have been through the query loop once to ensure that
// results are available.
if (queryTime) {
    glGetQueryObjectui64vEXT(presentTimeID,
        GL_QUERY_RESULT_ARB, &presentTime);
    glGetQueryObjectuivARB(presentDurationID,
        GL_QUERY_RESULT_ARB, &durationTime);

    float latency = (presentTime - sendTime[cur_query]) *
        .000001;
    float presentationInterval = (presentTime -
        lastPresentTime) * .000001;
    int bufsQueued = (int)(latency / presentationInterval);

    fprintf(stderr, "send time: %I64d    present time: %I64d
latency: %f msec    present interval: %f msec    bufs queued: %d
duration: %d frame\n",
        sendTime[cur_query], presentTime, latency,
        presentationInterval, bufsQueued, durationTime);

    lastPresentTime = presentTime;
}

// Query send time
glGetVideoui64vNV(1, GL_CURRENT_TIME_NV, &sendTime[cur_query]);

// Draw to video
glPresentFrameKeyedNV(1, 0,
    presentTimeID, presentDurationID,
    GL_FRAME_NV,
    GL_TEXTURE_RECTANGLE_NV, gTO, 0,
    GL_NONE, 0, 0);

if (cur_query == NUM_QUERIES) {
    cur_query = 0;
    queryTime = GL_TRUE;
}

```

## 8.2. Detecting Duplicate Frames

A duplicate frame will occur on the SDI output when a new frame is not ready in the queue at the time of the vertical retrace. This will happen when an application's draw time exceeds the time period between subsequent vertical retrace events on the outgoing SDI video signal. When a new frame is not sent prior to the next vertical retrace, one of two possible scenarios takes place. If, there is a frame already queued, then that frame is sent and the number of queued buffers is reduced by one. In this case, there will be no repeated frame on the SDI video output. In the case that the internal buffer queue is empty, the last sent frame will be scanned out again by the SDI video device. This will display as a duplicate frame.

**Note:** Internally, the Quadro FX SDI device only displays complete interlaced or progressively scanned frames. As such, only complete frames are displayed. In the case of interlace video, this behavior prevents the intermixing of fields from different video frames.

### 8.2.1. Using the GLX/WGL\_video\_out Extension

When an application uses the WGL\_NV\_video\_out or GLX\_NV\_video\_out extensions, dropped frames or fields or missed vsync events are detected by catching unexpected results in the value of the vsync counter `pulCounterOutputVideo` returned `wglGetVideoInfoNV()` or `glXGetVideoInfoNV()`. When these functions are called each time in the draw loop, the value of `pulCounterOutputVideo` should increment by 1 in the case of a progressive video format or when rendering fields in an interlaced video format and by 2 when rendering frames in an interlaced video format. An example of checking for dropped frames within the draw loop is outlined in Code Listing 28.

#### Code Listing 28: Detecting a Dropped/Duplicate Frame

```
unsigned long l_ulNumVertBlanks;
unsigned long l_ulNumBufs;
static unsigned long l_ulLastNumBufs = 0;
static unsigned long l_ulLastSent = 0;
static unsigned long l_ulLastNumVertBlanks = 0;

wglGetVideoInfoNV(ghpDevList[0], &l_ulNumBufs,
                  &l_ulNumVertBlanks);

if (gbInterlaced) {
    if (!options.field) {
        l_iBufType = WGL_VIDEO_OUT_FRAME;
        l_iVsyncDiff = 2;
        l_bBlock = options.block;
    } else {
        l_iBufType = l_bField1 ? WGL_VIDEO_OUT_FIELD_1 :
                        WGL_VIDEO_OUT_FIELD_2;
        l_bField1 = l_bField1 ? 0 : 1;
        l_bBlock = options.block; //l_bField1 ? FALSE : TRUE;
    }
}
```

```

        l_iVsyncDiff = 1;
    }
} else {
    l_iBufType = WGL_VIDEO_OUT_FRAME;
    l_iVsyncDiff = 2;
    l_bBlock = options.block;
}

wglSendPbufferToVideoNV(hpBuf, l_iBufType, &gBufCount, l_bBlock);

if ((l_ulNumVertBlanks != l_ulLastNumVertBlanks) &&
    (l_ulNumVertBlanks != (l_ulLastNumVertBlanks +
                          l_iVsyncDiff)))
    fprintf(stderr, "Warning:Dropped/Duplicate Frame\n");

l_ulLastNumVertBlanks = l_ulNumVertBlanks;

```

When in the case of a progressive video format, or when rendering fields, the vsync counter increments by more than one during subsequent execution of the draw loop, then a vertical retrace was missed. When rendering frames in an interleaved video format, if the vsync counter increments by more than two during subsequent execution of the draw loop, then a vertical retrace was missed. Now, to know if a duplicated frame or field is actually displayed on the SDI video output, this information must be used in coordination with the number of queued buffers as determined in Code Listing 26.

## 8.2.2. Using the GL\_present\_video Extension

When an application uses the GL\_NV\_present\_video extension, a query object is utilized to determine if a frame is duplicated on the SDI output. This is demonstrated in Code Listing 27.



## Chapter 9. Advanced Topics

This chapter outlines the use of some advanced features of the Quadro FX SDI.

---

### 9.1. Working with Two Video Channels

The Quadro FX SDI, in addition to operating in a dual-link configuration, can also be programmed to output the same video signal or two different video signals on the two video output jacks.

#### 9.1.1. Dual-Link Operation

The SMPTE standard defines dual-link video data formats. These formats utilize two video outputs in combination to deliver the complete video signal to the receiving device. The Quadro FX SDI software driver automatically recognizes these dual-link video formats when they are specified by an application and splits the resulting data across the two video channels. No special programming is required by the application to operate in this mode.

#### 9.1.2. Two Independent Video Channels

The video outputs of the Quadro FX SDI can also be configured to output two independent video data signals. In this operating mode, an application must first configure the Quadro FX SDI for dual outputs. To do this, the SDI device should be configured similarly to the single channel operating mode except for the data format. In this case, the data format must be set to one of the dual formats as shown in Code Listing 29.

##### Code Listing 29: Configuring the SDI Device to Output Two Independent Video Channels

```
l_gvoConfig.dataFormat=  
    NVGVODATAFORMAT_DUAL_R8G8B8_TO_DUAL_YCRCB422;
```

Once configured, an application must utilize separate render targets for each video channel. The application draw loop would then bind and update each render target in turn prior to utilizing `glPresentFrameDualFillNV()` to send the rendered data to the SDI device as demonstrated in Code Listing 30.

### Code Listing 30: Sending Two Independent Video Channels

```
gFB01.bind(gWidth, gHeight);
drawChannel1();
gFB01.unbind();

gFB02.bind(gWidth, gHeight);
drawChannel2();
gFB02.unbind();

glPresentFrameDualFillNV(1, 0, 0, 0, GL_FRAME_NV,
    GL_RENDERBUFFER_EXT, gFB01.renderbufferIds[0],
    GL_NONE, 0,
    GL_RENDERBUFFER_EXT, gFB02.renderbufferIds[0],
    GL_NONE, 0);
```

---

## 9.2. Sending the Desktop to Video Output

In addition to sending the contents of one or more pbuffers, buffer objects or textures as video output data, the desktop, or a region of the desktop can be displayed on the SDI video output from within an application. This mode of operation permits an application to control the Quadro FX SDI output in much the same way as the control panel.

The programming of this operating mode is similar to that described in Chapter 4 except that this usage model requires the use of the function `NvGvoDesktopOpen()` rather than `NvGvoOpen()`. Once the desktop is opened, the video output device is configured similarly to the buffer object case as outlined in code sample 4 except that in the desktop case, an application can initialize the `NVGVOOUTPUTREGION` structure to define the region of the desktop area that will be visible on the video output. Initialization of the Quadro FX SDI for desktop output to video from within an application is outlined in Code Listing 31. This example sends a  $1280 \times 720$  rectangular area of the desktop with the top left corner at location `(gX, gY)` to the SDI output as 720p5994 SDI video output.

## Code Listing 31: Configuring Desktop Video Output.

```

// Open the desktop for video output
if (NvGvoDesktopOpen(nAdapterNumber, NULL, NVGVOCLASS_SDI,
    NVGVO_O_WRITE_EXCLUSIVE, &g_hGVO) != NV_OK) {
    return E_FAIL;
}

// Configure video output parameters
NVGVOCONFIG l_gvoConfig;
l_gvoConfig.cbSize = sizeof(l_gvoConfig);
l_gvoConfig.dwFields = NVGVOCONFIG_SIGNALFORMAT |
    NVGVOCONFIG_DATAFORMAT |
    NVGVOCONFIG_OUTPUTREGION;

// Video output signal format
l_gvoConfig.signalFormat = NVGVOSIGNALFORMAT_720P_5994_SMPTE274;

// Video output data format
l_gvoConfig.dataFormat = NVGVODATAFORMAT_R8G8B8_TO_YCRCB422;

// Desktop region to output
NVGVOOUTPUTREGION l_gvoOutputRegion;
l_gvoOutputRegion.x = gX;
l_gvoOutputRegion.y = gY;
l_gvoOutputRegion.width = 1280;
l_gvoOutputRegion.height = 720;

// Set configuration
if (NvGvoConfigSet(g_hGVO, &l_gvoConfig) != NV_OK) {
    // Handle error
}

```

---

### 9.3. Colorspace Conversion

The Quadro FX SDI automatically performs ITU Rec. 601 or 709 colorspace conversion from RGB to YCrCb when required by the specified data format. This colorspace conversion is performed according to the following formula:

$$\begin{aligned}
 Y &= \text{offset}_Y + \text{scale}_Y * (r_y * R + g_y * G + b_y * B) \\
 Cb &= \text{offset}_{cb} + \text{scale}_{cb} * (r_{cb} * R + g_{cb} * G + b_{cb} * B) \\
 Cr &= \text{offset}_{cr} + \text{scale}_{cr} * (r_{cr} * R + g_{cr} * G + b_{cr} * B)
 \end{aligned}$$

where **scale** controls the magnitude of the resulting output range and **offset** controls the position of the result within that range. These values should be within -1.0 and 1.0.

The Quadro FX SDI processes the input values at 12-bit precision. In the case of 8-bit input data, the data is shifted up by 4 bits and the top 4 MSB bits are copied to the least 4 LSB bits. For 16-bit per-component input data, only the upper 12 bits are processed. The RGB data values come from the GPU output. The coefficient, offset and scale values are determined as described in the following sections.

### 9.3.1. Coefficients

The coefficients as well as the offsets in the formulas above are signed 16-bit integers which represent the values from -1.999 to +1.999. Each coefficient effectively has 14 bits of precision which should be adequate to provide high precision. All multiplications and additions are performed without truncation. Truncation only occurs at the final stage at which time 10- or 12-bit output values are extracted.

Table 9-1. SD ITU 601 Coefficients Table

	<b>r</b>	<b>g</b>	<b>b</b>
<b>Y</b>	0.2989	0.5865	0.1150
<b>C<sub>b</sub></b>	-0.1684	-0.3310	0.5000
<b>C<sub>r</sub></b>	0.5000	-0.4181	-0.08095

Table 9-2. HD ITU 709 Coefficients Table

	<b>r</b>	<b>g</b>	<b>b</b>
<b>Y</b>	0.2130	0.7156	0.0723
<b>C<sub>b</sub></b>	-0.1146	-0.38450	0.5000
<b>C<sub>r</sub></b>	0.5000	-0.4535	-0.0455

Custom coefficients may also be programmed via the API as outlined in Code Listing 32 and Code Listing 33. For applications that wish to scale the results within a particular range, the scale values must be considered in the computation of the coefficient values.

### 9.3.2. Scale

By default, the Quadro FX SDI performs colorspace conversion into what is commonly referred to as **Video Range**. In the 10-bit case, Y ranges [64 – 940] and CrCb [64 – 960]. The default *scale* values are then computed as follows.

$$scale_y = (940 - 64) / 1024 = 0.85547$$

$$scale_{cr} = (960 - 64) / 1024 = 0.875$$

$$scale_{cb} = (960 - 64) / 1024 = 0.875$$

In the 8-bit case, Y ranges from 16 – 235 while CrCb ranges from 16 – 240. The default **scale** values for **Video Range** are then similar:

$$\begin{aligned} \text{scale}_y &= (235 - 16) / 256 = 0.85547 \\ \text{scale}_{cr} &= (240 - 16) / 256 = 0.875 \\ \text{scale}_{cb} &= (240 - 16) / 256 = 0.875 \end{aligned}$$

To perform colorspace conversion into what is frequently referred to as **Film Range** or **Full Range** a custom colorspace conversion must be specified via the API. In the case of 10-bit full range, Y and CrCb ranges [4,1019]. The scale values are then computed as follows:

$$\begin{aligned} \text{scale}_y &= (1019 - 4) / 1024 = 0.992 \\ \text{scale}_{cr} &= (1019 - 4) / 1024 = 0.992 \\ \text{scale}_{cb} &= (1019 - 4) / 1024 = 0.992 \end{aligned}$$

### 9.3.3. Offset

The value of **offset<sub>y</sub>** is calculated as the ratio of the minimum value of luma within the range to the maximum value of luma within the range. In the case of 10-bit **Video Range**, **offset<sub>y</sub>** calculated as follows:

$$\text{offset}_y = (64 / 1024) = 0.0625$$

Meanwhile, the values of **offset<sub>cr</sub>** and **offset<sub>cb</sub>** are calculated as the middle point of the range divided by the maximum value in the range. Therefore, in the case of the 10-bit **Video Range**, the following calculation is used:

$$\text{offset}_{cr} = \text{offset}_{cb} = ((64 + 960) / 2) / 1024 = 0.5$$

Another way to think about **offset<sub>cr</sub>** and **offset<sub>cb</sub>** is as the value required to move the minimum result from the matrix multiplication and scale into the range of [0-1].

In the case of **Full Range** the 10-bit offset values are then computed as follows:

$$\begin{aligned} \text{offset}_y &= (4 / 1019) = 0.003925417 \\ \text{offset}_{cr} = \text{offset}_{cb} &= ((4 + 1019) / 2) / 1024 = 0.5 \end{aligned}$$

An application can override this default colorspace conversion behavior as outlined in Code Listing 32 for Windows and Code Listing 33 for Linux. However, applications should be aware that the final SDI output color range is always clamped to the following legal values.

**8-bit Range:** 4 – 251  
**10-bit Range:** 4 – 1019  
**12-bit Range:** 16 – 4079



## Code Listing 32: Specifying a Custom Colorspace Conversion on Windows.

```

.
.
.
// Colorspace Conversion
if (gbCSC) {
    l_gvoConfig.dwFields |= NVGVOCONFIG_CSCOVERRIDE;
    l_gvoConfig.bCSCOverride = TRUE;
    l_gvoConfig.dwFields |= NVGVOCONFIG_COLORCONVERSION;
    l_gvoConfig.colorConversion.colorOffset[0] = 0.0625;
    l_gvoConfig.colorConversion.colorOffset[1] = 0.5;
    l_gvoConfig.colorConversion.colorOffset[2] = 0.5;
    l_gvoConfig.colorConversion.colorScale[0] = 0.85547;
    l_gvoConfig.colorConversion.colorScale[1] = 0.875;
    l_gvoConfig.colorConversion.colorScale[2] = 0.875;
    l_gvoConfig.colorConversion.bCompositeSafe = TRUE;

    switch(geVideoFormat) {

        // ITU 709
        case VIDEO_FORMAT_1080P:
        case VIDEO_FORMAT_1080I:
        case VIDEO_FORMAT_720P:
            l_gvoConfig.colorConversion.colorMatrix[0][0] = 0.2130f;
            l_gvoConfig.colorConversion.colorMatrix[0][1] = 0.7156f;
            l_gvoConfig.colorConversion.colorMatrix[0][2] = 0.0725f;
            l_gvoConfig.colorConversion.colorMatrix[1][0] = 0.5000f;
            l_gvoConfig.colorConversion.colorMatrix[1][1] = -0.4542f;
            l_gvoConfig.colorConversion.colorMatrix[1][2] = -0.0455f;
            l_gvoConfig.colorConversion.colorMatrix[2][0] = -0.1146f;
            l_gvoConfig.colorConversion.colorMatrix[2][1] = -0.3850f;
            l_gvoConfig.colorConversion.colorMatrix[2][2] = 0.5000f;
            break;

        // ITU 601
        case VIDEO_FORMAT_487I:
        case VIDEO_FORMAT_576I:
            l_gvoConfig.colorConversion.colorMatrix[0][0] = 0.2991f;
            l_gvoConfig.colorConversion.colorMatrix[0][1] = 0.5870f;
            l_gvoConfig.colorConversion.colorMatrix[0][2] = 0.1150f;
            l_gvoConfig.colorConversion.colorMatrix[1][0] = 0.5000f;
            l_gvoConfig.colorConversion.colorMatrix[1][1] = -0.4185f;
            l_gvoConfig.colorConversion.colorMatrix[1][2] = -0.0810f;
            l_gvoConfig.colorConversion.colorMatrix[2][0] = -0.1685;
            l_gvoConfig.colorConversion.colorMatrix[2][1] = -0.3310f;
            l_gvoConfig.colorConversion.colorMatrix[2][2] = 0.5000f;
            break;

    } // switch
} else {
    l_gvoConfig.dwFields |= NVGVOCONFIG_CSCOVERRIDE;
    l_gvoConfig.bCSCOverride = FALSE;
} // if

.
.
.

```

```

if (NvGvoConfigSet(g_hGVO, &l_gvoConfig) != NV_OK) {

    // Handle error.

}

```

Once an application specifies a custom colorspace conversion matrix for a given data format, that matrix remains in effect until it is redefined, or at which point `bCSCOverride` is set to **FALSE** indicating that the default colorspace conversion for the given data format should be utilized.

### Code Listing 33: Specifying a Custom Colorspace Conversion on Linux.

```

// Setup color CSC matrix and offsets
if (op->csc) {
    switch(op->video_format) {
        case VIDEO_FORMAT_487I:
        case VIDEO_FORMAT_576I:
            colorMat[0][0] = 0.2991f;
            colorMat[0][1] = 0.5870f;
            colorMat[0][2] = 0.1150f;
            colorMat[1][0] = 0.5000f;
            colorMat[1][1] = -0.4185f;
            colorMat[1][2] = -0.0810f;
            colorMat[2][0] = -0.1685f;
            colorMat[2][1] = -0.3310f;
            colorMat[2][2] = 0.5000f;
            break;

        case VIDEO_FORMAT_720P:
        case VIDEO_FORMAT_1080I:
            colorMat[0][0] = 0.2130f;
            colorMat[0][1] = 0.7156f;
            colorMat[0][2] = 0.0725f;
            colorMat[1][0] = 0.5000f;
            colorMat[1][1] = -0.4542f;
            colorMat[1][2] = -0.0455f;
            colorMat[2][0] = -0.1146f;
            colorMat[2][1] = -0.3350f;
            colorMat[2][2] = 0.5000f;
            break;
    } // switch

    colorOffset[0] = 0.0625;
    colorOffset[1] = 0.5;
    colorOffset[2] = 0.5;
    colorScale[0] = 0.0625;
    colorScale[1] = 0.875;
    colorScale[2] = 0.875;

    XNVCTRLSetGvoColorConversion(dpy, screen, colorMat,
                                  colorOffset, colorScale);

    XNVCTRLSetAttribute(dpy, screen, 0,

```

```

        NV_CTRL_GVO_OVERRIDE_HW_CSC,
        NV_CTRL_GVO_OVERRIDE_HW_CSC_TRUE);
    } else {
        XNVCTRLSetAttribute(dpy, screen, 0,
            NV_CTRL_GVO_OVERRIDE_HW_CSC,
            NV_CTRL_GVO_OVERRIDE_HW_CSC_FALSE);
    }

```

## 9.4. Full-Scene Antialiasing

Full-scene antialiasing (FSAA) is required when computer-generated images are utilized in video and broadcast applications in order to remove the staircase artifacts caused by high-frequency transitions commonly found in point-sampled images. FSAA is implemented using a multisampling technique where the GPU uses multiple color samples to calculate the final pixel color. An application can utilize multisampling with both pbuffers and buffer objects as described in the next section.

### 9.4.1. Pbuffer Multisampling

An application sending pbuffers to the SDI device with the `WGL_video_out` or `GLX_video_out` OpenGL extensions that wishes to apply FSAA to the SDI video output must request a multisampled pixel format for pbuffer creation. Once a multisampled pbuffer is created, multisampling must be enabled within OpenGL. These steps are outlined in Code Listing 34.

#### Code Listing 34: Requesting a Multisampled Pixel Format During Pbuffer Creation.

```

// Request multisampled pixel format.
int attribList = {
    WGL_RED_BITS_ARB, 8,
    WGL_GREEN_BITS_ARB, 8,
    WGL_BLUE_BITS_ARB, 8,
    WGL_ALPHA_BITS_ARB, 8,
    WGL_STENCIL_BITS_ARB, 8,
    WGL_DEPTH_BITS_ARB, 24,
    WGL_SAMPLE_BUFFERS_ARB, GL_TRUE,
    WGL_SAMPLES_ARB, num_samples,
    WGL_DRAW_TO_PBUFFER_ARB, true,
    WGL_BIND_TO_VIDEO_RGBA_NV, true,
    0 };

wglChoosePixelFormat(hWindowDC, attribList, NULL, 1, &format, &nformats);

.
.
.

// Enable multisampling.
if (gbFSAA) {
    glEnable(GL_MULTISAMPLE_ARB);
}

```

In Code Listing 34, `num_samples`, can be specified as 1, 2, 4, 8 or 16, the number of desired samples per pixel. The greater the number, the more samples per pixel and improved image quality. However, the application programmer should be aware, that increasing the number of samples per pixel increases the video memory requirements of the application and in some cases, especially in the cases of HD video formats, the capacity of video memory may be exceeded. This may lead to reduced performance caused by swapping of textures and other video data to main memory, or pbuffer creation may fail due to lack of available video memory.

When the multisampled pbuffer is sent to the SDI video device with either `wglSendPbufferToVideoNV()` or `glxSendPbufferToVideoNV()`, the driver will automatically do the downsample blit prior to the SDI scanout.

For more information on multisampling, refer to the `GL_ARB_multisample` extension specification.

## 9.4.2. Multisampling with Buffer Objects

An application that wishes to use multisampling with buffer objects will utilize the capabilities enabled by the `GL_EXT_framebuffer_multisample` OpenGL extension. Unlike in the case of pbuffers, the application must perform the down sample bit and filter operation prior to sending the buffer objects to the SDI device.

The first step is to create an additional multisampled buffer object during OpenGL initialization. Frame buffer object creation is identical to that shown in Code Listing 6, except for the specification of render buffer storage as demonstrated in Code Listing 35.

### Code Listing 35: Requesting Multisample Render Buffer Storage.

```
if (num_samples > 1) {
    glRenderbufferStorageMultisampleEXT(GL_RENDERBUFFER_EXT,
                                        num_samples, texFormat,
                                        width, height);
} else {
    glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT, texFormat,
                            width, height);
}
```

Then, in the draw loop, the Code Listing 36 performs the down sample and filter from the multisample buffer object to the normal non-multisample buffer object prior to presenting the final buffer object to the SDI video device.

## Code Listing 36: Buffer Object Blit and Down Sample

```

// Bind buffer object
if (options.fsaa == 1)
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, gFBO);
else
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, gFBOMultiSampled)

    glEnable(GL_MULTISAMPLE);
// Draw frame content here
.
.
.

if (options.fsaa == 1){

    // Unbind FBO
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

} else {

    // If using multisample render buffer,
    // then blit to downsample and filter
    glBindFramebufferEXT(GL_READ_FRAMEBUFFER_EXT,
                          gFBOMultiSampled);
    glBindFramebufferEXT(GL_DRAW_FRAMEBUFFER_EXT,
                          gFBO);

    glBlitFramebufferEXT(0, 0, gWidth, gHeight,
                          0, 0, gWidth, gHeight,
                          GL_COLOR_BUFFER_BIT |
                          GL_DEPTH_BUFFER_BIT |
                          GL_STENCIL_BUFFER_BIT,
                          GL_NEAREST);

    glDisable(GL_MULTISAMPLE);

    // Unbind FBOs
    glBindFramebufferEXT(GL_READ_FRAMEBUFFER_EXT, 0);
    glBindFramebufferEXT(GL_DRAW_FRAMEBUFFER_EXT, 0);

    // Present final frame to the video device
    .
    .
    .
}

```

## 9.5. Calculating Video Memory Usage

When creating any graphics application, it is important to consider the graphics or video memory requirements of that application. Graphics or video memory is the physical memory that is located on the graphics card. Once the graphics memory footprint of an application exceeds the available memory on the card, the driver can no longer allocate buffers, textures and other graphics objects. In some cases, like in the case of pbuffer allocation, a memory allocation will simply fail within an application, in other cases, graphics objects will be swapped to system memory leading to reduced application performance. Table 9-3 demonstrates the calculation of the amount of video memory required by an application. This example represents a typical usage scenario for 1080i HD output with a  $1920 \times 1080$  visible frame buffer, and a single  $4\times$  multisampled  $1920 \times 1080$  pbuffer.

**Note:** The SDI device allocates 5 internal FP16 (2 bytes per component) buffers. These buffers permit an application to render up to five frames or fields ahead of the SDI scanout in the case when `wglSendPbufferToVideoNV()` and `glXSendPbufferToVideoNV()` are specified not to block.

Table 9-3. Video Memory Required by an Application

Framebuffer	Width:	1920	Height:	1080		
Color : (32-bit RGBA, double buffered)		1920 x 1080 x 4 x 2 / 1024 / 1024 =			15.82 MB	
Depth : (32-bit with packed stencil)		1920 x 1080 x 4 / 1024 / 1024 =			7.91 MB	
Overlay : (16-bit front + back)		1920 x 1080 x 2 x 2 / 1024 / 1024=			7.91 MB	
Total:					31.64 MB	
Pbuffer	Width:	1920	Height:	1080	Samples Per Pixel:	4
Color: (32-bit RGBA, double buffered)		1920 x 1080 x 4 x 4 x 2 / 1024 / 1024 =			63.28 MB	
Depth: (32-bit with packed stencil)		1920 x 1080 x 4 x 4 / 1024 / 1024 =			31.64 MB	
Total:					94.92 MB	
SDI Video	Width:	1920	Height:	1080	Num Buffers:	5
		1920 x 1080 x 8 x 5 / 1024 / 1024 =			79.10 MB	
Grand Total:					205.66 MB	

## 9.6. Working with Greater than 8 Bits Per-Component

SMPTE specifications support the transmission of greater than 8-bit per-component data as an SDI video signal. Applications wishing to transmit greater than 8-bit per-component data may do so by sending FP16 buffer objects or pbuffers to the Quadro FX SDI. FP16 is native GPU hardware format with a signed floating point format with a 10-bit mantissa and a 5-bit exponent. After a FP16 render target has been obtained, the renderbuffer, texture object or pbuffer must be filled with 16-bit floating point data prior to transfer. In some cases, this may require that the application image data be converted from integer to floating point. The data format specified in the video device configuration determines the expected render target type.

## 9.7. Data Integrity Check

The Quadro FX SDI provides the capability to test the integrity of the data cable between the graphics card and the SDI daughter board. When this mode is enabled, the SDI daughter card compares the color value of each pixel on a line to the color of the first pixel on that line and returns a count of the number of mismatched pixels. Due to the serial structure of the data cable, this test is designed to catch cases where the quality of the cable introduces errors into the data stream. An application that wishes to utilize this mode to verify the integrity of the data cable should display a test pattern that draws pixels of the identical color on each output line and be structured in the following form:

**Note:** This feature is only available on the Quadro FX 4500, Quadro FX 5500, Quadro FX 4600 and Quadro FX 5600 SDI.

### Code Listing 37: Basic Code Structure for Data Integrity Check.

```
void runDataIntegrityCheck()
{
    // Initialize video device
    initializeVideo();

    // Initialize OpenGL state
    initializeGL();

    // Draw initial frames
    for (int I = 0; I < 30; i++) {
        drawPattern();
    }

    // Enable data integrity checking
    enableDataCheck();

    // Draw remaining frames
    for (int i = 0; i < gNumFrames; i++) {
        drawPattern();
    }
}
```

```

    // Check errors
    checkErrors();

    // Disabled data integrity checking
    disableDataCheck();

    // Draw last frame
    drawPattern();

    // Cleanup OpenGL state
    cleanupGL;

    // Release video device.
    cleanupVideo();
}

```

The data integrity check mode is a hardware state enabled and disabled by calling `NvGvoConfigSet()` much the same way as other video control parameters are set by an application. Examples of functions that demonstrate the enabling and disabling of the data integrity check are shown in Code Listing 38.

### Code Listing 38: Enabling and Disabling the Data Integrity Check on Windows.

```

//
// Enable data integrity check
//
BOOL
enableDataCheck(GLvoid)
{
    NVRESULT l_nvResult = NV_OK;
    NVGVOCONFIG l_gvoConfig;

    l_gvoConfig.cbSize = sizeof(NVGVOCONFIG);
    l_gvoConfig.dwFields = 0;
    l_gvoConfig.dwFields = NVGVOCONFIG_DATAINTEGRITYCHECK;
    l_gvoConfig.bEnableDataIntegrityCheck = TRUE;

    l_nvResult = NvGvoConfigSet(g_hGVO, &l_gvoConfig);
    if ( l_nvResult != NV_OK ) {
        // Handle error
        return E_FAIL;
    }
    return S_OK;
}

//
// Disable data integrity check
//
BOOL
disableDataCheck(GLvoid)
{
    NVRESULT l_nvResult = NV_OK;
    NVGVOCONFIG l_gvoConfig;

```



```

    l_gvoConfig.cbSize = sizeof(NVGVOCONFIG);
    l_gvoConfig.dwFields = 0;
    l_gvoConfig.dwFields = NVGVOCONFIG_DATAINTEGRITYCHECK;
    l_gvoConfig.bEnableDataIntegrityCheck = FALSE;

    l_nvResult = NvGvoConfigSet(g_hGVO, &l_gvoConfig);
    if ( l_nvResult != NV_OK ) {
        // Handle error
        return E_FAIL;
    }
    return S_OK;
}

```

## 9.8. Composite Sync Termination

The Quadro FX SDI also provides the capability to enable and disable termination of the composite sync signal by calling `NvGvoConfigSet()` the same way that other video control parameters are set within an application. Examples of functions that demonstrate the enabling and disabling of composite sync termination are shown in Code Listing 39.

**Note:** This feature is only available on the Quadro FX 4500, Quadro FX 5500, Quadro FX 4600 and Quadro FX 5600 SDI.

### Code Listing 39: Enabling and Disabling Composite Sync Termination on Windows.

```

//
// Enable composite sync termination
//
BOOL
enableSyncTermination(GLvoid)
{
    NVRESULT l_nvResult = NV_OK;
    NVGVOCONFIG l_gvoConfig;

    l_gvoConfig.cbSize = sizeof(NVGVOCONFIG);
    l_gvoConfig.dwFields = 0;
    l_gvoConfig.dwFields = NVGVOCONFIG_COMPOSITETERMINATE;
    l_gvoConfig.bCompositeTerminate = TRUE;

    l_nvResult = NvGvoConfigSet(g_hGVO, &l_gvoConfig);
    if ( l_nvResult != NV_OK ) {
        // Handle error
        return E_FAIL;
    }
    return S_OK;
}

//

```

```

// Disable composite sync termination
//
BOOL
disableSyncTermination(GLvoid)
{
    NVRESULT l_nvResult = NV_OK;
    NVGVOCONFIG l_gvoConfig;

    l_gvoConfig.cbSize = sizeof(NVGVOCONFIG);
    l_gvoConfig.dwFields = 0;
    l_gvoConfig.dwFields = NVGVOCONFIG_COMPOSITETERMINATE;
    l_gvoConfig.bCompositeTerminate = FALSE;

    l_nvResult = NvGvoConfigSet(g_hGVO, &l_gvoConfig);
    if ( l_nvResult != NV_OK ) {
        // Handle error
        return E_FAIL;
    }
    return S_OK;
}

```

## 9.9. Specifying the Internal Buffer Queue Length

The driver software for the Quadro FX SDI maintains a collection of internal buffers. The default number of buffers is five. This internal buffer queue permits applications that do not set the `bBlock` argument to **TRUE** in `wglSendPbufferToVideoNV()` or `glXSendPbufferToVideoNV()` to render and send frames or fields ahead of when they will be scanned out by the SDI video device. On Windows XP, the number of internal buffers can be queried using `NvGvoConfigGet()` as outlined in Code Listing 40.

### Code Listing 40: Querying the Number of Internal Buffers on Windows.

```

//
// Get flip queue length
//
BOOL
getFlipQueueLength()
{
    // Get signal format from GVO config.
    NVGVOCONFIG l_gvoConfig = {0};
    l_gvoConfig.cbSize = sizeof(NVGVOCONFIG);
    l_gvoConfig.dwFields = NVGVOCONFIG_FLIPQUEUELENGTH;
    if (NvGvoConfigGet(g_hGVO, &l_gvoConfig) != NV_OK) {
        MessageBox(NULL, "Unable to get flip queue length.\n",
"Error", MB_OK);
        return FALSE;
    }
    return TRUE;
}

```

At the time that the video output device is configured and prior to calling either `wglBindVideoImageNV()` or `glXBindVideoImageNV()`, this number of internal buffers can be specified by an application using `NvGvoConfigSet()`. The number of internal buffers specified must be between two and seven.

#### Code Listing 41: Setting the Bumber of Internal Buffers on Windows.

```
//
// Set flip queue length
//
BOOL
setFlipQueueLength(unsigned int uiFQL)
{
    NVRESULT l_nvResult = NV_OK;

    // Get signal format from GVO config.
    NVGVOCONFIG l_gvoConfig = {0};
    l_gvoConfig.cbSize = sizeof(NVGVOCONFIG);
    l_gvoConfig.dwFields = NVGVOCONFIG_FLIPQUEUELENGTH;
    l_gvoConfig.dwFlipQueueLength = 2;
    l_nvResult = NvGvoConfigSet(g_hGVO, &l_gvoConfig);
    if ( l_nvResult != NV_OK ) {
        // Handle error
    }
    return TRUE;
}
```

On Linux, the number of internal buffers can be specified with `XNVCTRLsetAttribute()` as demonstrated below.

```
// Set number of internal buffers

XNVCTRLSetAttribute(dpy, screen, 0, NV_CTRL_GVO_FLIP_QUEUE_SIZE, 2);
```

### **Notice**

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

### **Trademarks**

NVIDIA, the NVIDIA logo and Quadro are trademarks or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

### **Copyright**

© 2008 NVIDIA Corporation. All rights reserved.

