NVIDIA CAPTURE SDK PROGRAMMING GUIDE

PG-06183-001_v06| June 2017

Programming Guide

DOCUMENT CHANGE HISTORY

PG-06183-001_v06

Version	Date	Authors	Description of Change
0.7	5/9/2011	BO	Initial draft
0.8	10/18/2011	JB	Adding information about NVFBC_TARGET_ADAPTER
0.9	1/2/2012	AC	Updated for GRID Toolkit version 1.1
0.91	7/20/2012	AC	Updated for Monterey Toolkit version 1.2
0.92	8/3/2012	JB	Removed NvEncodeAPI.dll references
1.0	5/21/2013	BO	Updated to match V2.0 of GRID SDK
2.1	7/29/2013	SD	Update to include GRID SDK V2.1 features
2.3	3/6/2014	AR	Update to include GRID SDK V2.3 features
3.0	7/8/2014	SD	Update for GRID SDK 3.0
4.0	5/12/2015	SD	Update for GRID SDK 4.0, include INVFBCHWEncoder interface, deprecated NVFBCTOH264HWEncoder interface.
4.1	7/9/2015	SD	Update to include NVFBCToDX9Vid interface, remove Tegra decode guide
4.1.1	11/17/2015	EY	Added section 2.9.1.6 with details on how to change the bitrate dynamically.
5.0	2/5/2016	SD	Update for NVIDIA Capture SDK 5.0, Added Section 2.11 to describe usage of difference maps, Added deprecation note for INVFBCHWEncoder interface
5.0	7/26/2016	SD	Update for NvFBCFrameGrabInfo::dwDriverInternalError diagnostic usage
6.0	1/20/2017	SD	Added information for the Capture SDK 6.0 release.
6.1	5/10/2017	SD	Update NvFBC DiffMap description

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1. OVERVIEW

The NVIDIA Capture Software Development Kit, previously called as GRID SDK is a comprehensive suite of tools for NVIDIA GPUs that enable high performance graphics capture and encoding. This Programming Guide describes how to use the various NVIDIA Capture SDK interfaces available on GRID, Quadro, and specific Tesla Products.

1.1 GPU ACCELERATED READBACK AND ENCODE

The NVIDIA Capture SDK includes two API interfaces for high performance readback of rendered content from the GPU and video encoding on the GPU:

1.1.1 NVFBC – NVIDIA Framebuffer Capture

The NVIDIA Framebuffer Capture (NVFBC) API captures and optionally compresses the entire Windows desktop or full-screen applications running on the supported Operating Systems (For list of Operating Systems, please refer to the SDK release notes). It essentially provides the same output as a real connected monitor to the GPU: a full desktop, with application windows, menu bar, composited overlay and hardware cursor. As such, NVFBC is ideally suited to *desktop capture and remoting*.

NVFBC has many advantages over existing methods of framebuffer capture. It is resilient to Aero DWM (enable/disable) changes and resolution changes. It operates asynchronously to graphics rendering because it is able to use the dedicated hardware compression and copy engines on the GPU. It delivers frame data to system memory faster than any other display output or other readback mechanisms all while having minimal impact on the rendering performance.

NVFBC is described in Chapter 2.

1.1.2 NVIFR – NVIDIA Inband Frame Readback

The NVIDIA Inband Frame Readback (NVIFR) API captures and optionally compresses an individual DirectX or OpenGL graphics render target. Unlike NVFBC, the output from NVIFR does not include any window manager decoration, composited overlay, cursor or taskbar; it solely provides the pixels rendered into the render target, as soon as their rendering is complete, ahead of any compositing that may be done by the windows manager. In fact, NVIFR does not require that the render target even be visible on the Windows desktop. It is ideally suited for *application capture and remoting*, where the output of a single application, rather than the entire desktop environment, is captured.

NVIFR is intended to operate *inband* with a rendering application, either as part of the application itself, or as part of a shim layer operating immediately below the application. Like NVFBC, NVIFR operates asynchronously to graphics rendering, using dedicated hardware compression and copy engines in the GPU, and delivering pixel data to system memory with minimal impact on rendering performance.

NVIFR is described in Chapter 3.

1.1.3 API Reference documents

Details of APIs, parameters, etc. are documented in the API reference documents "***NVFBC.chm***" and "***NVIFR.chm***" that are installed with the NVIDIA Capture SDK.

The term "***NVFBC API Reference document***" used in this programming guide refers to ***NVFBC.chm***.

The term "***NVIFR API Reference document***" used in this programming guide refers to ***NVIFR.chm***.

Chapter 2. NVFBC – FRAMEBUFFER CAPTURE

NVIDIA Framebuffer Capture (NVFBC) is a high performance, low latency API for reading back display frames from one or more GPU *display heads*. NVIDIA GPUs typically support at least two display heads, and these are usually associated with a physical display output such as a DVI, DisplayPort, or HDMI connector. NVFBC provides essentially the same output one would see on a monitor connected to the GPU: a full desktop, with application windows, menu bar, composited overlay and hardware cursor. By operating asynchronously to graphics rendering and using dedicated hardware compression and copy engines in the GPU, NVFBC delivers frame data to CPU-based applications faster than any other display output or readback mechanism, with minimal impact on rendering performance.

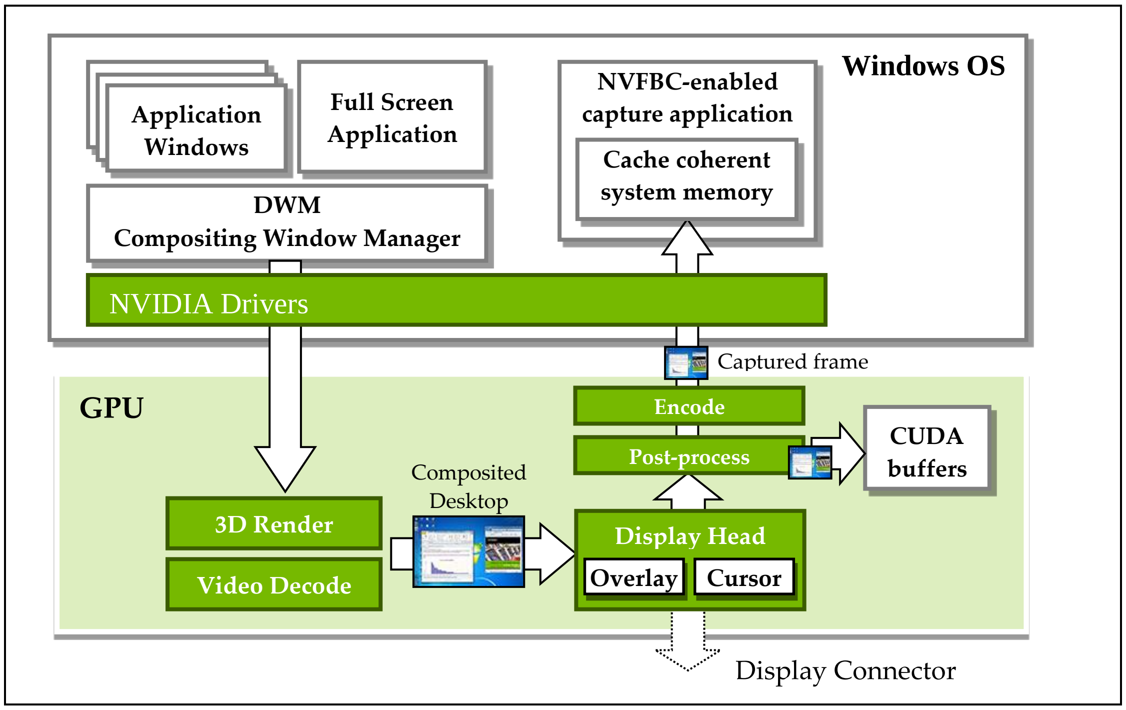


Figure 1 NVFBC framebuffer capture

NVFBC is supported on the Windows 7, Windows 8, Windows 8.1 & Windows 10 operating systems, and offers these features:

- Automatic capture of on-screen updates (graphics updates or mouse moves)
- Full operation through screen resolution changes, and Windows Aero on/off
- Compositing of hardware cursor and overlay with the base desktop image
- Color space conversion
- Cropping and scaling
- Pixel and tile-based differencing
- Stereoscopic capture
- Output of H.264 or H.265 compressed frames into cache-coherent, pinned system memory
- Output of uncompressed frames into cache-coherent, pinned system memory; this mode is ideally suited for use with CPU-based post-processing / compression implementations.
- Output of uncompressed frames into D3D9-mapped buffers in the GPU framebuffer; this mode is ideally suited for use with D3D9 post-processing / compression implementations[1].
- Output of uncompressed frames into CUDA-mapped buffers in the GPU framebuffer; this mode is ideally suited for use with CUDA-based post-processing / compression implementations[2]

Operation of NVFBC is straightforward: after doing one-time setup of the NVFBC API on application load, an application creates an *NVFBC object* for each GPU display head it wishes to read back from, and then enters a processing loop on each NVFBC object to read back frames from each head. Figure 2 provides an overview of the processing flow, which is described in more detail in the following sections.

Note:

[1]*This mode works with baremetal, direct attached GPUs, and all vGPU profiles. This is the recommended path when using vGPU profiles that support two or more virtual machines sharing a single GPU. For such vGPU profiles, the CUDA driver is not available. We recommend using this NvFBC path so that capture and encode can be fully accelerated.*

[2]*This mode is supported in baremetal, direct attached GPUs, and vGPU profiles that limit one virtual machine. The CUDA driver is available and supported in this configuration.*

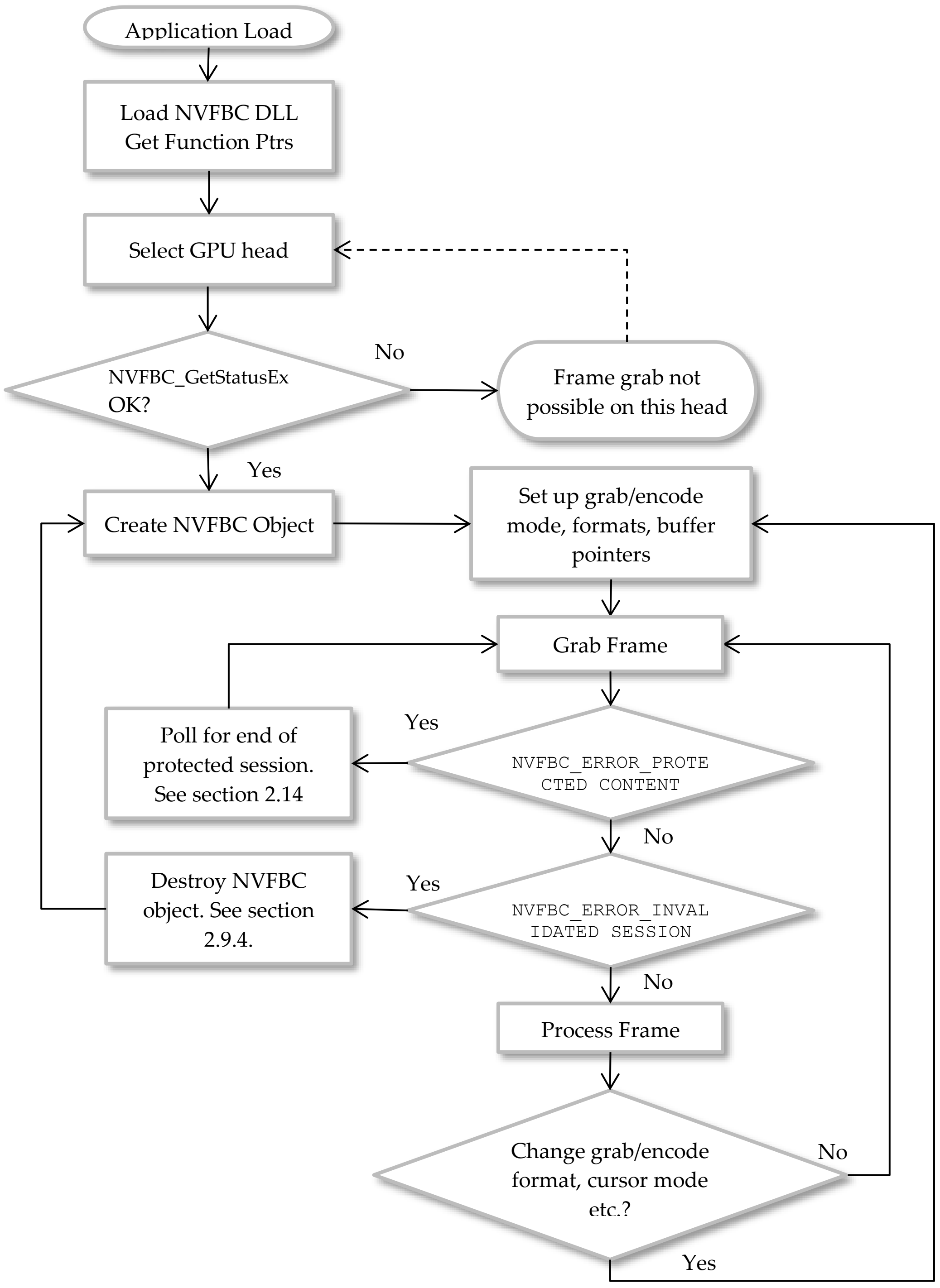

Application Load
Load NVFBC DLL
Get Function Ptrs
Select GPU head
NVFBC_GetStatusEx
OK?
No
Frame grab not possible on this head
Yes
Create NVFBC Object
Set up grab/encode mode, formats, buffer pointers
Grab Frame
Poll for end of protected session. See section 2.14
Yes
NVFBC_ERROR_PROTECTED_CONTENT
No
Destroy NVFBC object. See section 2.9.4.
Yes
NVFBC_ERROR_INVALIDATED_SESSION
No
Process Frame
Change grab/encode format, cursor mode etc.?
No
Yes

Figure 2 Overview of NVFBC application flow

2.1 HEADER FILES AND CODE SAMPLES

This manual provides an overview of how to use NVFBC. Further details are contained in the NVFBC header files and code samples that are included in the NVIDIA Capture SDK Toolkit:

NVFBC header files are installed in `%CAPTURESDK_PATH%\inc\NVFBC`. All NVFBC applications should include one or more of the mode-specific NVFBC header files, depending on the functionality desired:

Header file	**Description**
`NVFBC.h`	Top level header file included by all NVFBC applications
`NVFBCToSys.h`	Defines ToSys interface; reads back uncompressed frames to system memory.
`NVFBCCuda.h`	Defines Cuda interface; reads back uncompressed frames to CUDA-mapped buffers in the GPU's framebuffer.
`NVFBCToDx9vid.h`	Defines the DX9Vid interface; reads back uncompressed frames to D3D9-mapped buffers in the GPU's framebuffer.
`NVFBCHWEnc.h`	Defines the capture+encode interface; reads back compressed video frames to system memory. Compression is performed using NVENC HW Encoder engine. Supports H.264 and HEVC compression.
`NVHWEnc.h`	Definitions for NVENC HW Encoder configuration settings, to be included with NVFBCHWEnc.h in the application.

Table 1 NVFBC header files

The following NVFBC code samples are installed in `%CAPTURESDK_PATH%\samples\`

Please refer to the NVIDIA Capture SDK Samples Description document for details about NVFBC samples included with the SDK.

2.2 PREPARING THE API FOR USE

Regardless of the mode in which an application uses the NVFBC API, the following initialization steps are required:

- Enable NVFBC Registry Settings (Every time after GRID GPU driver update)
- Load the NVFBC DLL (At application load time)
- Obtain NVFBC function pointers (At application load time)

2.2.1 Programmatically Enabling\Disabling NVFBC

GRID SDK 3.1 adds the ability for an NVFBC client to programmatically Enable\Disable NVFBC without needing to perform Steps mentioned in section 2.2.1 separately.

After loading the NVFBC DLL, the application should obtain a pointer to the NVFBCEnable() API. The following code snippet demonstrates how to enable NVFBC.

```
// Load NVFBC function pointer
pfnNVFBC_Enable = (NVFBC_EnableFunctionType)
            GetProcAddress(handleNVFBC, "NVFBC_Enable");
// Check NVFBC Status
NVFBCStatusEx status;
pfnNVFBC_GetStatusEx(&status);
if (!status.bCurrentlyCapturing && !status.bIsCapturePossible)
{
    // NVFBC Capture has not been enabled. Try Enabling it.
    NVFBCRESULT res = pfnNVFBC_Enable(NVFBC_STATE_ENABLE);
}
```

The following code snippet demonstrates how to disable NVFBC.

```
// Check NVFBC Status
NVFBCStatusEx status;
pfnNVFBC_GetStatusEx(&status);
if (!status.bCurrentlyCapturing && status.bIsCapturePossible)
{
    // NVFBC Capture has been enabled, no capture process is currently
active. It is safe to disable NVFBC.
    NVFBCRESULT res = pfnNVFBC_Enable(NVFBC_STATE_DISABLE);
}
```

This API needs administrator privileges to work correctly. The API will return NVFBC_ERROR_INSUFFICIENT_PRIVILEGES in case it is not called from a process that has Administrator privileges.

2.2.2 Enabling NVFBC using Registry Settings

The NVFBC technology requires a registry key to be set before the related functionality can be accessed by an application. NVFBC object creation will fail if the registry settings are not enabled.

The procedure to enable these registry settings is described in section 4.1.3.

An alternate method to enable NVFBC is described in section 2.2.1

Please note that this is required to be done each time there is an update to the GRID GPU driver installed on the system where NVFBC API is being used.

2.2.3 Loading the DLL

The NVFBC API is accessed via a 32- or 64- bit dynamic link library (DLL), which must be loaded by the application before calling any NVFBC functions:

```
// 32-bit application

HINSTANCE handleNVFBC = ::LoadLibrary("NVFBC.dll");

// 64-bit application

HINSTANCE handleNVFBC = ::LoadLibrary("NVFBC64.dll");
```

Note: The NVFBC DLLs are located in `the NVIDIA driver directory`. When shipping an application that uses the NVIDIA Capture SDKs, you do not need to ship these dlls, as they are included in the driver. See Chapter 4 for further guidance on shipping GRID-enabled applications.

2.2.4 Accessing NVFBC function pointers

After loading the NVFBC DLL, the next step is to get pointers to the `NVFBC_GetStatusEx()`, `NVFBC_CreateEx()`, and `NVFBC_SetGlobalFlags()` functions in the DLL. This is accomplished with calls to `GetProcAddress()`:

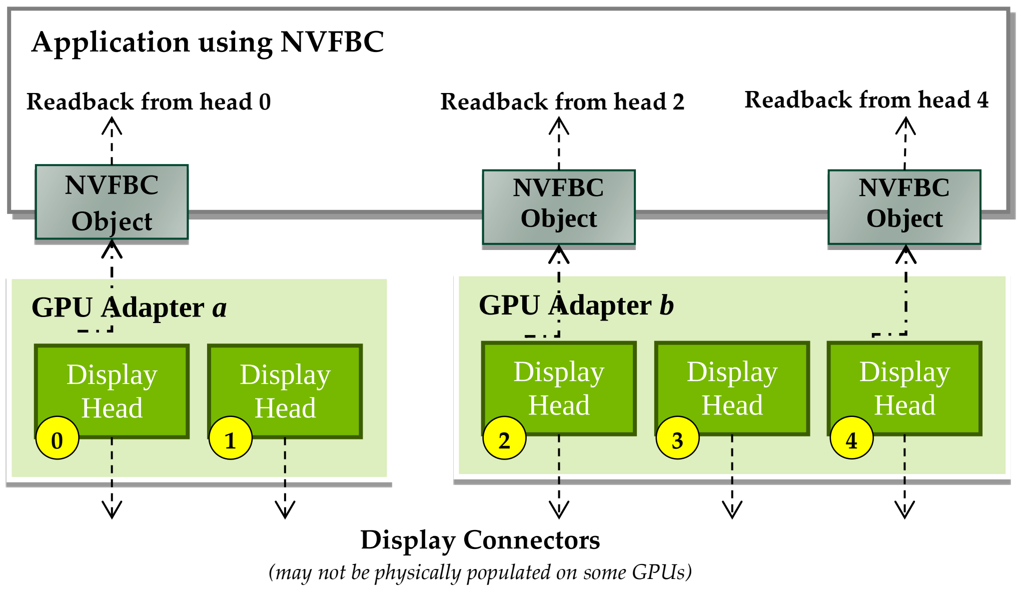
```
// Load NVFBC function pointers
pfnNVFBC_GetStatus = (NVFBC_GetStatusExFunctionType)
            GetProcAddress(handleNVFBC, "NVFBC_GetStatusEx");

pfnNVFBC_Create = (NVFBC_CreateFunctionExType)
            GetProcAddress(handleNVFBC, "NVFBC_CreateEx");

pfnNVFBC_SetGlobalFlags = (NVFBC_SetGlobalFlagsType)
            GetProcAddress(handleNVFBC, "NVFBC_SetGlobalFlagsEx");

pfnNVFBC_Enable = (NVFBC_EnableFunctionType)
            GetProcAddress(handleNVFBC, "NVFBC_Enable");
```

2.3 SELECTING A GPU HEAD FOR READBACK

The NVFBC API reads back frames from one or more GPU *display heads*. Exactly one display head can be associated with an NVFBC session object. This association needs to be established while creating the NVFBC object, and it stays bound throughout the session lifetime. Note that exactly one NVFBC session can be associated with a display at any given time, making this a 1:1 association.

NVIDIA GPUs typically support at least two display heads, and there may be multiple NVIDIA GPUs present in the system.

Figure 3 shows an example system with two NVIDIA GPUs, each with multiple display heads. An application is reading back frames from one display head on the first GPU, and two display heads on the second GPU, and has created three NVFBC objects for this purpose.



n Direct3D9 ordinal adapter ID: set `NVFBC_TARGET_ADAPTER` to this value at NVFBC object creation time, to select the head to be associated with the object.

Figure 3 NVFBC objects association with GPU display heads

Display heads are numbered using the ordinal adapter identifier value assigned by Direct3D9. The number of attached D3D9 adapters can be acquired by calling `IDirect3D9::GetAdapterCount()`. To get information about specific adapters `IDirect3D9::GetAdapterMonitor()` or `IDirect3D9::GetDeviceCaps()` should be called.

Note: For detailed code samples showing how to enumerate adapters using Direct3D9 and cross-reference with those returned by GDI, and how to enable display heads that do not have monitors attached to them, see the white paper *Displayless Multi-GPU on Windows 7*, which is included with the GRID Toolkit

The client can select a GPU head for readback by setting up NVFBC using either of the following methods:

- Initialize a D3D9 device using Direct3D9 ordinal adapter identifier of the display head to be read back and pass the IDirect3DDevice9 object to `NVFBCCreateEx` as `NVFBCCreateParams::pDevice`
- If the client does not want to manage a D3D9 device, the client should set `NVFBCCreateParams::pDevice` to `NULL` and Direct3D9 ordinal adapter identifier of the display head to be read back should be passed to NVFBCCreateEx as `NVFBCCreateParams::dwAdapterIdx`

2.4 VERIFYING NVFBC STATUS

Once a display head has been selected, verify the status of the NVFBC interface for that head by calling `NVFBC_GetStatusEx()`, with the parameter `NVFBCStatusEx::dwAdapterIdx` set to the selected adapter ordinal:

```
NVFBCStatusEx status;
...
// Get NVFBC status

pfnNVFBC_GetStatusEx(&status);
```

Please refer to the API reference document for details regarding the `NVFBC_GetStatusEx()` API.

It is safe to call `NVFBC_GetStatusEx()` multiple times to poll for detecting completion of NVFBC enable\disable operation.

2.5 CREATING NVFBC OBJECTS

All NVFBC readback operations are exposed as methods in NVFBC classes. Distinct classes are used to support the different readback modes supported by NVFBC (to system memory, to CUDA buffers, and H.264 encode). After using `NVFBC_GetStatusEx()` to verify that readback is possible on a GPU head; the next step is to create an NVFBC class object associated with the GPU head.

An NVFBC object is associated with exactly one GPU display head. Selecting a head for readback is described in section 2.3

Note: At most one NVFBC object can be active on a display head at any given time.

NVFBC objects cannot be created while any application is currently running in fullscreen mode on any head. NVFBC-enabled applications should typically create NVFBC objects for available display heads at system intialization time, before any applications run in full screen mode.

To create an NVFBC object, allocate variables to store the maximum display width and height, then call `NVFBC_CreateEx()` to create the object. The example below creates an `NVFBC_TO_SYS` object, to read back data directly to system memory, but the create call is similar for all classes of NVFBC object:

```
NVFBCRESULT (NVFBCAPI * NVFBC_CreateFunctionExType)
(void * pCreateParams);

// Example of usage:

// NVFBC_TARGET_ADAPTER env variable previously set...
NVFBCCreatParams createParams   = {0};
createParams.dwVersion          = NVFBC_CREATE_PARAMS_VER.
createParams.dwInterfaceType    = NVFBC_TO_SYS;
createParams.dwMaxDisplayWidth  = -1;
createParams.dwMaxDisplayHeight = -1;
createParams.pDevice            = pD3DDevice; //Pointer to app's
D3DDevice
createParams.pPrivateData       = NULL;
createParams.dwPrivateDataSize  = 0;
createParams.dwInterfaceVersion = NVFBC_DLL_VERSION;

createParams.pNVFBC = NULL; //OUT, pointer to requested NVFBC object

NVFBCRESULT result;
result = pfnNVFBC_CreateEx(&createParams);
```

The CreateParams.dwInterfaceType parameter specifies the type of NVFBC object to be created;

`dwCaptureType` value	Notes
`NVFBC_TO_SYS`	Reads back frames to locked, cache-coherent buffers in system memory. See section 2.5.2.
`NVFBC_SHARED_CUDA`	Reads back frames in ARGB format to CUDA-mapped buffers resident in the GPU's framebuffer. See section 0.
`NVFBC_SHARED_CUDA_YUV420P`	Reads back frames in YUV420p format to CUDA-mapped buffers resident in the GPU's framebuffer. See section 0.
`NVFBC_TO_HW_ENCODER`	Reads back compressed video frames to locked, cache-coherent buffers in system memory. See section 2.9.

Table 2 NVFBC capture types

The `createParams.dwMaxDisplayWidth` and `createParams.dwMaxDisplayHeight` parameters are used to return a maximum supported resolution supported by the NVFBC interface.

The `createParams.pPrivateData` argument is reserved for future use, and should be passed as `NULL`.

If successful, the `NVFBCCreateEx()` call returns `NVFBC_SUCCESS`, with a pointer to a newly-created NVFBC object in CreateParams.pDevice. Otherwise the call returns `an` error code, as enumerated in NVFBC.h as `NVFBCRESULT`. An error can be caused by:

- An NVFBC object already active for the head indicated by `NVFBC_TARGET_ADAPTER`.
- An application running in full screen mode on any display head.

2.5.1 Maximum supported resolution

The maximum supported display resolution returned from the `NVFBC_CreateEx()` call is a static property of the NVFBC interface, and is deliberately larger than the typical maximum resolution supported on a single GPU display head. This allows NVFBC to internally pre-allocate readback buffers, and handle dynamic resolution changes during capture/readback without needing to reallocate buffers.

Similarly, applications using NVFBC may wish to use the reported maximum resolution to size and pre-allocate any data buffers they use for handling readback data.

2.5.2 Frame grab info structure

Information about the grabbed frame is returned in the `NVFBCFrameGrabInfo` structure passed to the call. Please refer to the API reference for details regarding `NVFBCFrameGrabInfo` members.

2.6 CAPTURING TO SYSTEM MEMORY

To capture uncompressed frames to system memory, create an NVFBC object from the NVFBCToSys class by specifying:

```
CreateParams.dwInterfaceType = NVFBC_TO_SYS;
```

Please refer to NVFBCToSys Object definition in the NVFBC API Reference document.

2.6.1 Setting up the NVFBCToSys object

Before frames can be grabbed, the NVFBCToSys object requires a setup call, `NVFBCToSysSetup()`, to specify the target capture mode and the required grab format. The `NVFBCToSysSetup()` method returns pointers to buffers that will contain readback data and difference maps. `NVFBCToSysSetup()` may subsequently be called again for an NVFBCToSys object, any time the application wishes to change the capture mode or grab format.

Please refer to NVFBC API reference document for details regarding `NVFBC_TOSYS_SETUP_PARAMS` struct.

```
NVFBCRESULT NVFBCToSysSetUp(NVFBC_TOSYS_SETUP_PARAMS *pParam)

//! Example of usage:

unsigned char *pBuffer = NULL;
unsigned char *pDiffMap = NULL;

NVFBC_TOSYS_SETUP_PARAMS setupParams = {0};
setupParams.dwVersion = NVFBC_TOSYS_SETUP_PARAMS_VER;
setupParams.bWithHWCursor = FALSE;
setupParams.bDiffMap = TRUE;
setupParams.eDiffMapBlockSize = (NvU32) NVFBC_TOSYS_DIFFMAP_BLOCKSIZE_128X128;
setupParams.eMode = NVFBC_TOSYS_ARGB;
setupParams.ppBuffer = &pBuffer;
setupParams.ppDiffMap = &pDiffMap;

NVFBCRESULT result;
result = NVFBCToSys->NVFBCToSysSetUp(&setupParams);
```

If successful, `NVFBCToSysSetup()` returns `NVFBC_SUCCESS`, and the object is now ready for frame grabbing. Otherwise it returns one of the errors enumerated in `NVFBCRESULT`.

2.6.1.1 Capture mode

The `NVFBC_TOSYS_SETUP_PARAMS::eMode` parameter is a capture mode that specifies the pixel format in which frame captures will be returned. Please refer the NVFBC API reference for details regarding `NVFBCToSysBufferFormat` enum.

2.6.1.2 Hardware cursor handling

The `NVFBC_TOSYS_SETUP_PARAMS::bWithHWCursor` parameter controls hardware cursor compositing: if specified as `TRUE`, any active hardware cursor is composited into read back frames, otherwise it is not. If a software cursor is active, this will be composited into the frame regardless of this parameter setting.

2.6.1.3 Readback buffer

The `NVFBC_TOSYS_SETUP_PARAMS::ppBuffer` parameter is a pointer to pointer to a buffer that will contain the read back frame data, in the format specified by the capture mode in `setupParams.eMode`. Note that NVFBC allocates the buffer for the frame data, the application simply passes a `void **` argument to receive a pointer to the NVFBC-allocated buffer.

2.6.1.4 Difference maps

The `NVFBC_TOSYS_SETUP_PARAMS::ppDiffMap` parameter is a pointer to a pointer to a buffer (allocated by NVFBC) that will contain a frame-to-frame difference map whenever a frame is read back. If your application will not make use of difference maps, this argument may be passed as `NULL`.

Please refer to section 2.6.2.3 for more information.

2.6.2 Grabbing frames with NVFBCToSys

To grab a frame with NVFBCToSys, call `NVFBCToSysGrabFrame()`:

```
NVFBCRESULT NVFBCToSysGrabFrame(NVFBC_TOSYS_GRAB_FRAME_PARAMS *pParam);

// Example of usage:

NVFBCFrameGrabInfo frameGrabInfo;

NVFBC_TOSYS_GRAB_FRAME_PARAMS grabFrameParams = {0};
grabFrameParams.dwVersion = NVFBC_TOSYS_GRAB_FRAME_PARAMS_VER;
grabFrameParams.dwFlags = NVFBC_TOSYS_NOFLAGS;
grabFrameParams.dwTargetWidth = -1;
grabFrameParams.dwTargetHeight = -1;
grabFrameParams.dwStartX = 0;
grabFrameParams.dwStartY = 0;
grabFrameParams.eGMode = NVFBC_TOSYS_SOURCEMODE_FULL;
grabFrameParams.pNVFBCFrameGrabInfo = &frameGrabInfo;

NVFBCRESULT result;
result = NVFBCToSys->NVFBCToSysGrabFrame(&grabFrameParams);
```

Please refer to NVFBC API Reference document for details regarding `NVFBC_TOSYS_GRAB_FRAME_PARAMS` struct.

By default, `NVFBCToSysGrabFrame()` is a blocking call, and returns once a new frame is available.

If the call is successful, `NVFBCToSysGrabFrame()` returns `NVFBC_SUCCESS`, information about the captured frame is returned in the `NVFBCFrameGrabInfo` struct passed as `NVFBC_TOSYS_GRAB_FRAME_PARAMS`::`pNvFBCFrameGrabInfo`, and the readback buffer obtained via the `NVFBCToSysSetup()` call contains the captured frame data. If a diffmap pointer was specified in `NVFBCToSysSetup()`, the diffmap buffer will contain a difference map, unless this is the first frame captured after the `NVFBCToSysSetup()` call, in which case the difference map will contain all bits set to '1'.

If there is an error, `NVFBCToSysGrabFrame()` returns one of the errors enumerated in `NVFBCRESULT`, indicating that a frame was not successfully captured. This can occur if the return value is:

- `NVFBC_ERROR_PROTECTED_CONTENT`: Protected content is currently being displayed. See section 2.14.1 for more information on handling protected content.
- The NVFBC object must be re-created. See section 2.14.2 for a discussion of the factors that require this.

Please refer to the NVFBC API Reference for details about `NVFBCRESULT` values.

2.6.2.1 Blocking and non-blocking frame grabs

The `NVFBC_TOSYS_GRAB_FRAME_PARAMS::dwFlags` parameter can be used to control whether NVFBC should perform a blocking frame grab or a non-blocking frame grab. Please refer to `NVFBC_TOSYS_GRAB_FLAGS` in the NVFBC API Reference document to know more about the supported values.

Note: under normal operation, Windows may update the display image on a head with exactly the same image that was previously displayed. In this case, a blocking call `NVFBCToSysGrabFrame()` will return a new frame with identical content to the previous one. Similarly, a non-blocking call to `NVFBCToSysGrabFrame()` will always return the latest frame, which may not have changed from the last time `NVFBCToSysGrabFrame()` was called. In both these cases, Difference Maps (when enabled) can be used to detect when the frame returned is identical to the previous one.

2.6.2.2 Scaling and cropping

The `NVFBC_TOSYS_GRAB_FRAME_PARAMS::eGMode` parameter specifies the grab mode for the frame capture, controlling cropping or scaling on the captured frame. The grab modes are mutually exclusive.

Please refer to `NVFBCToSysGrabMode` in the NVFBC API Reference document to know more about supporting scaling\cropping modes.

Scaling of the capture frame may be useful when remoting frames to a client that has a lower resolution than the local frame resolution. By downscaling at the point of capture, the amount of data compressed and transmitted to the remote client is reduced.

Cropping of the frame may be useful when supporting a “panning” mode on a remote client with lower resolution than the local frame, or to capture a specific application window’s output. (Note that NVIFR, described in Chapter 3, allows for direct capture of a DirectX render context ahead of any window manager compositing, and is typically better suited for capture/remoting of specific application windows.)

2.6.2.3 Difference Maps

The difference maps feature is available with NVFBCToSys and NVFBCToDx9Vid interfaces. Please refer to Section 2.11 for details.

2.6.3 Grabbing Mouse Separately with NVFBCToSys

Cursor Capture support is available across all NVFBC interfaces starting from GRID SDK 4.1 and associated GPU driver.

Refer to Section 2.10 for details.

2.6.4 Releasing the NVFBCToSys object

After you have finished using `NVFBCToSys` you must release it to properly free the resources.

```
NVFBCRESULT NVFBCToSysRelease();
// Example code
toSys->NVFBCtoSysRelease();
```

2.7 CAPTURING TO CUDA DEVICE MEMORY

To capture uncompressed frames to CUDA mapped buffers, create an NVFBC object from the NVFBCCuda class by specifying `NVFBC_SHARED_CUDA` in the `NVFBC_CreateEx()` call. Please refer to NVFBCCuda Object definition in the NVFBC API Reference document.

NVFBCCuda *pNVFBCCuda = (NVFBCCuda*)
  pfnNVFBC_CreateEx(NVFBC_SHARED_CUDA, &maxDisplayWidth,
    &maxDisplayHeight, NULL);

`NVFBC_SHARED_CUDA` is used to capture frames in 32-bit ARGB pixel format, one byte per channel.

2.7.1 Allocating a CUDA device buffer

NVFBCCuda requires the caller to allocate buffers to hold grabbed frames. Use NVFBCCuda's `NVFBCCudaGetMaxBufferSize()` method to determine the maximum-sized frame that NVFBC can grab, then use `cudaMalloc()` to allocate a buffer in the GPU's framebuffer to accommodate it.

Please refer to `NVFBCCudaGetMaxBufferSize()` in NVFBC API Reference document.

```
// Determine maximum size that NVFBC can return

DWORD maxBufferSize = pNVFBCCuda->NVFBCCudaGetMaxBufferSize();
...

// Use CUDA driver API to alloc memory on CUDA device for grabbed frame

CUdeviceptr buffer;
cuMemAlloc(&buffer, maxBufferSize);
...

// Or, using the CUDA runtime API:
void * buffer;

cudaMalloc(&buffer, maxBufferSize);
```

2.7.2 Grabbing frames with NVFBCCuda

Grabbing a frame with NVFBCCuda is a single step process – call `NVFBCCudaGrabFrame()` to trigger a readback, supplying a previously allocated CUDA buffer. Please refer to `NVFBCCudaGrabFrame` in the NVFBC API Reference document.

```
NVFBCRESULT NVFBCCudaGrabFrame (NVFBC_CUDA_GRAB_FRAME_PARAMS *pParams)
// Example of usage:

NVFBCFrameGrabInfo frameGrabInfo;

NVFBC_CUDA_GRAB_FRAME_PARAMS grabParams = {0};
grabParams.dwVersion = NVFBC_CUDA_GRAB_FRAME_PARAMS_VER;
grabParams.pCUDADeviceBuffer = (void *)buffer;
grabParams.pNVFBCFrameGrabInfo = &frameGrabInfo;
grabParams.dwFlags = NVFBC_TOCUDA_NOFLAGS;

NVFBCRESULT result;
result = pNVFBCCuda->NVFBCCudaGrabFrame(&grabParams);
```

The input/output parameters to `NVFBCCudaGrabFrame()` are described in the following sections. By default, `NVFBCCudaGrabFrame()` is a blocking call, and returns once a new frame is available. If the call is successful, `NVFBCCudaGrabFrame()` returns `NVFBC_SUCCESS`, information about the captured frame is returned in the `NVFBC_CUDA_GRAB_FRAME_PARAMS::pNVFBCFrameGrabInfo` structure, and the supplied buffer contains the captured frame data.

If there is an error, `NVFBCCudaGrabFrame()` returns one of the errors enumerated in `NVFBCRESULT`, indicating that a frame was not successfully captured. This can occur if the return value is:

- `NVFBC_ERROR_PROTECTED_CONTENT`: Protected content is currently being displayed . See section 2.14.1 for more information on handling protected content.

- `NVFBC_ERROR_INVALIDATED_SESSION`: The NVFBC object must be re-created (`bMustRecreate` is `TRUE` in the frame grab info structure). See section 2.14.2 for a discussion of the factors that require this.

Please refer to the NVFBC API Reference for details about `NVFBCRESULT` values.

2.7.2.1 Blocking and non-blocking frame grabs

The `NVFBC_CUDA_GRAB_FRAME_PARAMS::dwFlags` parameter specifies miscellaneous control flags for the call. Please refer to `NVFBC_CUDA_FLAGS` in the NVFBC API Reference document for more details.

Note: under normal operation, Windows may update the display image on a head with exactly the same image that was previously displayed. In this case, a blocking call NVFBCCudaGrabFrame() will return a new frame with identical content to the previous one. Similarly, a non-blocking call to NVFBCCudaGrabFrame() will always return the latest frame, which may not have changed from the last time NVFBCCudaGrabFrame() was called

2.7.2.2 Frame grab info structure

Information about the grabbed frame is returned in the `NVFBCFrameGrabInfo` structure passed as `NVFBC_CUDA_GRAB_FRAME_PARAMS::pNVFBCFrameGrabInfo`.Please refer to the NVFBC API Reference document for details regarding `NVFBCFrameGrabInfo`.

2.8 CAPTURING TO IDIRECT3DSURFACE9* BUFFERS

To capture uncompressed frames to system memory, create an NVFBC object from the NVFBCToSys class by specifying:

```
CreateParams.dwInterfaceType = NVFBC_TO_DX9_VID;
```

Please refer to `NvFBCToDx9Vid` Object definition in the NVFBC API Reference document.

2.8.1 Setting up the NVFBCToDx9Vid object

Before frames can be grabbed, the `NvFBCToDx9Vid` object requires a setup call, `NvFBCToDx9VidSetUp()`, to specify the target capture mode and the required grab format. The `NvFBCToDx9VidSetUp()` method registers client provided d3d9 surfaces for use with NVFBC.

Please refer to NVFBC API reference document for details regarding `NVFBC_TODX9VID_SETUP_PARAMS` struct.

```
NVFBCRESULT NvFBCToDx9VidSetUp(NVFBC_TODX9VID_SETUP_PARAMS *pParam);

// Example of usage:
NVFBCRESULT res = NVFBC_SUCCESS;
NVFBC_TODX9VID_SETUP_PARAMS setup = {0}NVFBC_TODX9VID_OUT_BUF
fbcOut[NBUFFERS] = {0};

//! Allocate surfaces
for (int nsurfout = 0; nsurfout < NBUFFERS; nsurfout++)
{
    //! Create a surface to pass to NVFBC
    hr = pD3DDev->CreateOffscreenPlainSurface(curWidth, curHeight,
                 D3DFMT_A8R8G8B8, D3DPOOL_DEFAULT,
                 &fbcOut[nsurfout].pPrimary, NULL);
    if (FAILED(hr))
    {
        fprintf(stderr, "Failed to allocate NVFBC output surface.\n");
        return E_FAIL;
    }
}

//! Configure the grabber, get grab output buffer handle.
setup.dwVersion = NVFBC_TODX9VID_SETUP_PARAMS_VER;
setup.bWithHWCursor = false;
setup.dwNumBuffers = NBUFFERS;
setup.eMode = NVFBC_TODX9VID_ARGB;
setup.ppBuffer = fbcOut;
setup.bWithHWCursor = true;
res = toDx9Vid->NvFBCToDx9VidSetUp(&setup);
```

If successful, `NVFBCToDx9VidSetup()` returns `NVFBC_SUCCESS`, and the object is now ready for frame grabbing. Otherwise it returns one of the errors enumerated in `NVFBCRESULT`.

2.8.1.1 Capture mode

The `setupParams.eMode` parameter is a capture mode that specifies the pixel format in which frame captures will be returned. Please refer the NVFBC API reference for details regarding `NVFBCToDx9VidBufferFormat` enum.

2.8.1.2 Hardware cursor handling

The `setupParams.bWithHWCursor` parameter controls hardware cursor compositing: if specified as `TRUE`, any active hardware cursor is composited into read back frames, otherwise it is not. If a software cursor is active, this will be composited into the frame regardless of this parameter setting.

2.8.1.3 Readback buffers

The application should create d3d9 surfaces of a supported pixel format, and pass them as an array using the parameter `NVFBC_TODX9VID_SETUP_PARAMS::ppBuffer`

The NVFBC API will register these surfaces as target surfaces for holding the grabbed images. A maximum of 3 output surfaces can be registered with a given NVFBC session. Calling this API again with new surfaces will instruct NVFBC to invalidate previous configuration and register the new surfaces.

The application is responsible for deallocating the surfaces, after releasing the NVFBC session or invalidating the registration with NVFBC.

2.8.2 Grabbing frames with NVFBCToDx9Vid

To grab a frame with NVFBCToDx9Vid, call `NVFBCToDx9VidGrabFrame()`:

```
NVFBCRESULT NVFBCToDx9VidGrabFrame(NVFBC_TODX9VID_GRAB_FRAME_PARAMS
*pParam);

// Example of usage:
NVFBCFrameGrabInfo frameGrabInfo;
NVFBC_TODX9VID_GRAB_FRAME_PARAMS

grabFrameParams = {0};
grabFrameParams.dwVersion = NVFBC_TODX9VID_GRAB_FRAME_PARAMS_VER;
grabFrameParams.dwFlags = NVFBC_TODX9VID_NOFLAGS;
grabFrameParams.eGMode = NVFBC_TOSYS_SOURCEMODE_FULL;
grabFrameParams.dwBufferIdx = i;
grabFrameParams.pNVFBCFrameGrabInfo = &frameGrabInfo;

NVFBCRESULT result;
result = toDx9Vid->NVFBCToDx9VidGrabFrame(&grabFrameParams);
```

Please refer to NVFBC API Reference document for details regarding `NVFBC_TODX9VID_GRAB_FRAME_PARAMS` struct.

By default, `NVFBCToDx9VidGrabFrame()` is a blocking call, and returns once a new frame is available.

If the call is successful, `NVFBCToDx9VidGrabFrame ()` returns `NVFBC_SUCCESS`, information about the captured frame is returned in the `NVFBCFrameGrabInfo` struct passed as `NVFBC_TODX9VID_GRAB_FRAME_PARAMS::pNVFBCFrameGrabInfo`, and one of the readback buffer registered via the `NVFBCToDx9VidSetup()` call contains the captured frame data.

If there is an error, `NVFBCToSysGrabFrame()` returns one of the errors enumerated in `NVFBCRESULT`, indicating that a frame was not successfully captured. This can occur if the return value is:

- `NVFBC_ERROR_PROTECTED_CONTENT`: Protected content is currently being displayed. See section 2.14.1 for more information on handling protected content.
- The NVFBC object must be re-created. Please refer section 2.14.2 for a discussion of the factors that require this.

Please refer to the NVFBC API Reference for details about `NVFBCRESULT` values.

2.8.2.1 Blocking and non-blocking frame grabs

The `NVFBC_TODX9VID_GRAB_FRAME_PARAMS::dwFlags` parameter can be used to control whether NVFBC should perform a blocking frame grab or a non-blocking frame grab. Please refer to `NVFBC_TODX9VID_GRAB_FLAGS` in the NVFBC API Reference document to know more about the supported values.

Note: under normal operation, Windows may update the display image on a head with exactly the same image that was previously displayed. In this case, a blocking call `NVFBCToDx9VidGrabFrame()` will return a new frame with identical content to the previous one. Similarly, a non-blocking call to `NVFBCToDx9VidGrabFrame ()` will always return the latest frame, which may not have changed from the last time `NVFBCToDx9VidGrabFrame ()` was called. In both these cases, Difference Maps (when enabled) can be used to detect when the frame returned is identical to the previous one.

2.8.2.2 Scaling and cropping

The `NVFBC_TODX9VID_GRAB_FRAME_PARAMS::eGMode` parameter specifies the grab mode for the frame capture, controlling cropping or scaling on the captured frame. The grab modes are mutually exclusive.

Please refer to `NVFBCToDx9VidGrabMode` in the NVFBC API Reference document to know more about supporting scaling\cropping modes.

Scaling of the capture frame may be useful when remoting frames to a client that has a lower resolution than the local frame resolution. By downscaling at the point of capture, the amount of data compressed and transmitted to the remote client is reduced.

Cropping of the frame may be useful when supporting a "panning" mode on a remote client with lower resolution than the local frame, or to capture a specific application window's output. (Note that NVIFR, described in Chapter 3, allows for direct capture of a DirectX render context ahead of any window manager compositing, and is typically better suited for capture/remoting of specific application windows.)

2.8.3 Releasing the NVFBCToDx9Vid object

After you have finished using `NVFBCToSys` you must release it to properly free the resources.

```
// Example code
NVFBCRESULT NVFBCToDX9VidRelease();
toDX9Vid->NVFBCToDX9VidRelease ();
```

2.9 CAPTURING WITH HARDWARE VIDEO COMPRESSION

To capture compressed video frames to system memory, create an NVFBC object from the `INVFBCHWEncoder` class by specifying `NVFBC_TO_HW_ENCODER` in the `NVFBC_CreateEx()` call. Please refer to `INVFBCHWEncoder` object definition in the NVFBC API Reference document.

Note: INVFBCHWEncoder interface is deprecated starting with NVIDIA Capture SDK 5.0. Future versions of the SDK will not include this interface definition. However, compatibility with applications that are already compiled with this interface will be preserved until further notice.

A combination of INVFBCToDX9Vid interface for capture and NVIDIA Video Codec SDK for encoding is recommended as a replacement.

```
INVFBCToHWEncoder *pNVFBCHWEnc = (INVFBCToHWEncoder *)
    pfnNVFBC_Create(NVFBC_TO_HW_ENCODER, &maxDisplayWidth,
                     &maxDisplayHeight, NULL);
```

This NVFBC interface is codec-agnostic, and supports H.264 as well as HEVC compression.

Note: Not all NVIDIA GPUs that support NVFBC support hardware compression. If the GPU does not support hardware compression, the `NVFBC_CreateEx()` call for an `NVFBC_TO_HW_ENCODER` object will return `NULL`.

Also, note that not all NVIDIA GPUs that support hardware compression can support HEVC compression. If the GPU does not support HEVC hardware compression, the `NVFBC_CreateEx()` call for an `NVFBC_TO_HW_ENCODER` object will return `NULL`.

2.9.1 Setting up the INVFBCToHWEncoder object

2.9.1.1 Checking HW Video Encoder Capabilities

`INVFBCToHWEncoder::NVFBCGetHWEncCaps()` can be used to check HW Video encoder capabilities like supported codec, rate control modes, encoding presets, etc. Please refer to the NVFBC API Reference document for details.

This should be done before calling `INVFBCToHWEncoder::NVFBCHWEncSetup()`.

2.9.1.2 Performing the HW Video Encoder Setup

Before frames can be grabbed, the `INVFBCToHWEncoder` object requires a setup call, `NVFBCHWEncSetup()`, to specify the capture mode and video encoding parameters.

Please refer NVFBC API Reference Document to know more about `NVFBC_HW_ENC_CONFIG_PARAMS`, `NVFBC_HW_ENC_SETUP_PARAMS`, `NVFBCHWEncSetUp()`, and related data types.

```
// Example of usage:

// Set the encoding parameters
NVFBC_HW_ENC_CONFIG_PARAMS = {0};
encodeConfig.dwVersion = NVFBC_HW_ENC_CONFIG_PARAMS_VER;
encodeConfig.eCodec = NV_HW_ENC_H264; // Use NV_HW_ENC_HEVC for HEVC
encodeConfig.dwProfile = 77;        // Main profile
encodeConfig.dwFrameRateNum = 30;
encodeConfig.dwFrameRateDen = 1;  // Set the target frame rate at 30
encodeConfig.dwAvgBitRate = 8000000; // Avg bitrate of 8 Mbps
encodeConfig.dwPeakBitRate = (NvU32)(8000000 * 1.50); // Set peak
```

```
                                                   // bitrate to 150%
                                                   // of average
encodeConfig.dwGopLength = 100; // The I-Frame frequency
encodeConfig.eRateControl = NVFBC_HW_ENC_PARAMS_RC_VBR;
encodeConfig.dwQP = 26; // Quantization parameter, between 0 and 51
encodeConfig.dwNumBFrames = 0; // Number of bi-directional frame refs.
encodeConfig.bOutBandSPSPPS = 0; // Use inband SPSPPS, if you need to
                                 // grab headers on demand use
                                 // outband SPSPPS
encodeConfig.bRecordTimeStamps = 0; // Don't record timestamps
encodeConfig.stereoFormat = NVFBC_HW_ENC_STEREO_NONE; // No stereo

NVFBC_HW_ENC_SETUP_PARAMS setupParams = {0};
setupParams.dwVersion = NVFBC_HW_ENC_SETUP_PARAMS_VER;
setupParams.bWithHWCursor = TRUE;
setupParams.EncodeConfig = encodeConfig;
// Setup the grab with hardware cursor and encode
NVFBCRESULT result;
result = pNVFBCHWEnc->NVFBCHWEncSetUp(&setupParams);
```

If successful, `NVFBCHWEncSetUp()` returns `NVFBC_SUCCESS`, and the object is now ready for frame grabbing. Otherwise, `NVFBCHWEncSetUp()` returns one of the errors enumerated in `NVFBCRESULT`.

Note: Not all NVIDIA GPUs that support hardware compression can support HEVC compression.

If the GPU does not support HEVC hardware compression, the `NVFBCHWEncSetUp()` call for `NVFBC_HW_ENC_CONFIG_PARAMS::eCodec = NV_HW_ENC_HEVC` will return `NULL`.

2.9.1.3 Selecting Video Compression Standard

The `NVFBC_HW_ENC_SETUP_PARAMS::eCodec` parameter controls the type of video compression that will be used. Please refer to `NV_HW_ENC_CODEC` enum in NVFBC API Reference document for details.

2.9.1.4 Hardware cursor handling

The `NVFBC_HW_ENC_SETUP_PARAMS::bWithHWCursor` parameter controls hardware cursor compositing: if specified as `TRUE`, any active hardware cursor is composited into captured frames prior to compression, otherwise it is not. If a software cursor is active, this will be composited into the frame regardless of this parameter setting.

2.9.1.5 Video encoding parameters

The NVFBC_HW_ENC_CONFIG_PARAMS structure defines various parameters used to configure the NVIDIA HW Video Encoder. Please refer to the NVFBC API Reference document for details.

2.9.1.6 Changing Bitrate Dynamically

The NV_HW_ENC_PIC_PARAMS structure includes flags to change the bitrate and encoding parameters dynamically. The structure requires that you modify the NVFBC_HW_ENC_GRAB_FRAME_PARAMS flag and the bitrate parameters below before the call to grab/encode the frame:

```
NV_HW_ENC_PIC_PARAMS EncodeParams
EncodeParams::bDynamicBitrate = 1;
EncodeParams::dwNewAvgBitrate       =  NEW_AVERAGE_BITRATE;
EncodeParams::dwNewPeakBitrate      =  NEW_PEAK_BITRATE;
EncodeParams::dwNewVBVBufferSize    =  NEW_VBV_BUFFER_SIZE;
EncodeParams::dwNewVBVInitialDelay  =  NEW_VBV_INITIAL_DELAY
```

2.9.2 Grabbing frames with INVFBCToHWEncoder

To grab a frame using INVFBCToHWEncoder, allocate an output buffer in system memory to hold the compressed bitstream, then call NVFBCHWEncGrabFrame().

2.9.2.1 Sizing and allocating the output buffer

Compressed video frames are written into caller-allocated buffers, typically allocated in cache-coherent system memory using malloc() or an equivalent Win32 function. A two megabyte buffer is appropriate for capture of 1920x1080 resolution.

2.9.2.2 Capturing frames

To grab a frame and capture as a video bitstream, call `NVFBCHWEncGrabFrame()`

`Please refer NVFBC API Reference document for details of the API, parameters, etc.`

```
// Example of usage:

NVFBC_HW_ENC_GRAB_FRAME_PARAMS fbcHwEncGrabFrameParams = {0};
NV_HW_ENC_GET_BIT_STREAM_PARAMS frameInfo = {0};

unsigned int dwOutputBufferSize = 2*1024*1024;
//! Setup a buffer to put the encoded frame in
outputBuffer = (unsigned char *)malloc(dwOutputBufferSize);

frameInfo.dwVersion = NV_HW_ENC_GET_BIT_STREAM_PARAMS_VER;
fbcHwEncGrabFrameParams.dwVersion = NVFBC_HW_ENC_GRAB_FRAME_PARAMS_VER;
fbcHwEncGrabFrameParams.dwFlags   = NVFBC_HW_ENC_NOWAIT;
fbcHwEncGrabFrameParams.NVFBCFrameGrabInfo = grabInfo;
fbcHwEncGrabFrameParams.GetBitStreamParams = frameInfo;
fbcHwEncGrabFrameParams.pBitStreamBuffer = outputBuffer;

//! Grab and encode the frame
res = encoder->NVFBCHWEncGrabFrame(&fbcHwEncGrabFrameParams);
```

The input/output parameters to `NVFBCHWEncGrabFrame()` are described in the NVFBC API Reference document.

By default, `NVFBCHWEncGrabFrame()` is a blocking call, and returns once a new frame is available.

If the call is successful, `NVFBCHWEncGrabFrame()` returns `NVFBC_SUCCESS`, information about the captured frame is returned in the `grabFrameParams.pNVFBCFrameGrabInfo` structure, and the caller-allocated buffer pointed at by `pBitStream` contains the compressed frame's bitstream. The size of the output bitstream in bytes is written to `NV_HW_ENC_GET_BIT_STREAM_PARAMS::dwByteSize` (i.e. `NVFBC_HW_ENC_GRAB_FRAME_PARAMS::GetBitStreamParams.dwByteSize)`.

If there is an error, `NVFBCHWEncGrabFrame()` returns one of the errors enumerated in `NVFBCRESULT`, indicating that a frame was not successfully captured and compressed. This can occur if the return value is:

- `NVFBC_ERROR_PROTECTED_CONTENT` : Protected content is currently being displayed. See section 2.14.1 for more information on handling protected content.
- `NVFBC_ERROR_INVALIDATED_SESSION`: The NVFBC object must be re-created (`bMustRecreate` is `TRUE` in the frame grab info structure). See section 2.14.2 for a discussion of the factors that require this.

2.9.2.3 Blocking and non-blocking frame grabs

The `NVFBC_HW_ENC_GRAB_FRAME_PARAMS::dwFlags` parameter can be used to control whether NVFBC should perform a blocking frame grab or a non-blocking frame grab. Please refer to `NVFBC_HW_ENC_GRAB_FLAGS` in the NVFBC API Reference document to know more about the supported values.

Note: under normal operation, Windows may update the display image on a head with exactly the same image that was previously displayed. In this case, a blocking call NVFBCHWEncGrabFrame() will grab and encode a new frame with identical content to the previous one. Similarly, a non-blocking call to NVFBCHWEncGrabFrame() will always grab and encode the latest frame, which may not have changed from the last time NVFBCHWEncGrabFrame() was called

2.9.2.4 Frame grab info structure

Information about the grabbed frame is returned in the `NVFBCFrameGrabInfo` structure passed as `NVFBC_HWENC_GRAB_FRAME_PARAMS::pNVFBCFrameGrabInfo`.

Please refer to the NVFBC API Reference document for details regarding `NVFBCFrameGrabInfo`.

2.9.2.5 NV_HW_ENC_GET_BIT_STREAM_PARAMS structure

Information about the compressed video frame is returned in the `NV_HW_ENC_GET_BIT_STREAM_PARAMS` structure passed in `NVFBC_HW_ENC_GRAB_FRAME_PARAMS::GetBitStreamParams`.

Please refer NVFBC API Reference document for details.

2.9.2.6 Scaling and cropping

The `NVFBC_HW_ENC_GRAB_FRAME_PARAMS::eGMode` parameter specifies the grab mode for the frame capture, controlling cropping or scaling on the captured frame. The grab modes are mutually exclusive.

Please refer to `NVFBC_HW_ENC_GRAB_MODE` in the NVFBC API Reference document to know more about supporting scaling\cropping modes.

Scaling of the capture frame may be useful when remoting frames to a client that has a lower resolution than the local frame resolution. By downscaling at the point of capture, the amount of data compressed and transmitted to the remote client is reduced.

Cropping of the frame may be useful when supporting a “panning” mode on a remote client with lower resolution than the local frame, or to capture a specific application window’s output. (Note that NVIFR, described in Chapter 3, allows for direct capture of a DirectX render content ahead of any window manager compositing, and is typically better suited for capture/remoting of specific application windows.)

2.9.3 Reading Sequence and Picture Parameter Sets

The SPS (sequence parameter set) and PPS (picture parameters set) headers contain data about the encoded frames in the stream, such as their resolution, that is necessary to decode them properly. Typically, a decoder will only need to decode these headers once, before decoding the first encoded frame in the scene.

If `NV_HW_ENC_CONFIG_PARAMS::bRepeatSPSPPSHeader` is set to TRUE(1), SPS and PPS headers will be returned with every I-frame.

If `NV_HW_ENC_CONFIG_PARAMS::bOutBandSPSPPS` parameter is set to FALSE (0), one set of SPS and PPS headers will be generated with the first I-frame in the stream. Subsequent I-frames will not include these headers, so the first encoded I-frame must be decoded in order to continue decoding the rest of the stream.

If this parameter is set to TRUE (1), then the SPS and PPS headers will not be returned with any encoded frame, but must be obtained by calling `NVFBCHwEncGetStreamHeader()`. After decoding these headers, a decoder should be able to decode any I-frame generated by the decoder and then decode the subsequent P- and B-frames. This will allow a decoder to successfully start decoding from intermediate points in the stream without having to decode the first encoded frame.

Use `NVFBCHWEncGetStreamHeader()` to read sequence and picture parameter sets for the current encode stream. Please refer NVFBC API Reference for details about parameters.

```
// Example of usage:
NvU32 size;
char * buffer = malloc(1024);
NVFBC_HW_ENC_GET_STREAM_HEADER_PARAMS getHeaderParams;
getHeaderParams.dwVersion = NVFBC_HW_ENC_GET_STREAM_HEADER_PARAMS_VER;
getHeaderParams.dwSize = size;
getHeaderParams.pBuffer = buffer;

NVFBCRESULT result;

result = pNVFBCHWEnc->NVFBCHWEncGetStreamHeader(&getHeaderParams);
```

`pBuffer` is a caller-allocated buffer used to return the parameter sets, and should be at least 1024 bytes in size. `pSize` is a pointer to an `NvU32` that returns the numbers of bytes written to `pBuffer` by the function.

If successful, `NVFBCHWEncGetStreamHeader()` returns `NVFBC_SUCCESS`, and `NVFBC_HW_ENC_GET_STREAM_HEADER::dwSize` indicates how many bytes were written. Otherwise `NVFBCHWEncGetStreamHeader()` returns one of the errors enumerated in `NVFBCRESULT`.

2.9.4 Releasing the INVFBCToHWEncoder object

After you have finished using `NVFBCToHWEncoder` you must release it to properly free the resources.

```
NVFBCRESULT NVFBCHWEncRelease();

// Example code
pNVFBCHWEnc->NVFBCHWEncRelease();
```

2.9.5 Using Intra-Refresh with INVFBCToHWEncoder

Intra-Refresh is an error-resiliency feature supported by NVIDIA Capture SDK, which allows the client to enable gradual decoder refresh or intra-refresh. This is supported only for GOP structures that do not use B-frames.

For example:

> If Intra-Refresh cycle count = n and no. of Macroblocks per frame = m, then for 1st frame, 0 to (m/n) - 1 macroblocks are coded as intra-predicted, for 2nd frame (m/n) to (2m/n) – 1 macroblocks are coded as intra-predicted, and so on.

To use this feature, the client should set

```
NV_HW_ENC_CONFIG_PARAMS::bEnableIntraRefresh = 1;
```

To initiate Intra-refresh, the client should follow this example:

```
NVFBC_HW_ENC_GRAB_FRAME_PARAMS fbcHwEncGrabFrameParams = {0};

NV_HW_ENC_GET_BIT_STREAM_PARAMS frameInfo = {0};
NV_HW_ENC_PIC_PARAMS encParams = {0};

unsigned int dwOutputBufferSize = 2*1024*1024;
//! Setup a buffer to put the encoded frame in
outputBuffer = (unsigned char *)malloc(dwOutputBufferSize);
frameInfo.dwVersion = NV_HW_ENC_GET_BIT_STREAM_PARAMS_VER;
encParams.dwVersion = NV_HW_ENC_PIC_PARAMS_VER:

//! Start an Intra-Refresh cycle over n frames.
encParams.bForceIntraRefresh = 1;
encParams.dwIntraRefreshCount = n;

fbcHwEncGrabFrameParams.dwVersion = NVFBC_HW_ENC_GRAB_FRAME_PARAMS_VER;
fbcHwEncGrabFrameParams.dwFlags    = NVFBC_HW_ENC_NOWAIT;
fbcHwEncGrabFrameParams.NVFBCFrameGrabInfo = grabInfo;
fbcHwEncGrabFrameParams.GetBitStreamParams = frameInfo;
fbcHwEncGrabFrameParams.pBitStreamBuffer = outputBuffer;

//! Grab and encode
res = encoder->NVFBCHWEncGrabFrame(&fbcHwEncGrabFrameParams);
```

Note that using Reference frame Invalidation in conjunction with Intra-Refresh is not supported. Reference frame invalidation requests will be ignored if client has enabled Intra-Refresh.

2.9.6 Using dynamic slice mode encoding with INVFBCToHWEncoder

Dynamic Slice mode encoding allows the client to configure how the encoded picture will be divided into slices. This is done by setting two parameters: `NV_HW_ENC_CONFIG_PARAMS::eSlicingMode` and `NV_HW_ENC_CONFIG_PARAMS::dwSlicingModeParam` before calling `NVFBCHWEncSetup()`.

Please refer `NV_HW_ENC_SLICING_MODE` in NVFBC API Reference for usage details.

2.9.7 INVFBCToHWEncoder Rate Control Modes

Please refer NVFBC API Reference document for details regarding supported Rate Control Modes.

2.9.8 Using Adaptive Quantization with INVFBCToHWEncoder

Adaptive Quantization (AQ) can be controlled using `NV_HW_ENC_CONFIG_PARAMS::bEnableAdaptiveQuantization` flag.

Quantization artifacts like blockiness in a flat region are more visible than in a complex region. With AQ enabled, the goal is to improve quality in a flat region. With Adaptive Quantization enabled, the quantization parameter is set depending upon complexity of macroblock data, thus assigning higher values of qp for macroblocks in high complex regions and lower values of qp for macroblocks in flat regions, thereby improving the visual quality of flat regions.

Note: Adaptive quantization works only with 2 pass rate control modes

2.9.9 Using Lossless encoding with INVFBCToHWEncoder

To retain the same quality after encoding-decoding, a lossless encoding feature is introduced. There is no loss of data in this mode; however, the size of the bit stream is large compared to lossy encoding.

Client should call NVFBCHWEncGetCaps() to check for `NV_HW_ENC_GET_CAPS::bLosslessEncodingSupported` for the currently set codec type before configuring lossless encode, as not all NVIDIA GPUs support lossless encoding.

```
//! Example:
//! Check capability
NV_HW_ENC_GET_CAPS caps = {0};
caps.dwVersion = NV_HW_ENC_GET_CAPS_VER;
caps.eCodec = codec;
pNVFBCHWEnc->NVFBCHWEncGetCaps(&caps);

NV_HW_ENC_CONFIG_PARAMS encodeConfig = {0};
//! Other encode config init
if (caps.bLosslessEncodingSupported)
{
    encodeConfig.dwProfile = 244;
    encodeConfig.ePresetConfig= NVFBC_HW_ENC_PRESET_LOSSLESS_HP;
    encodeConfig.eRateControl = NVFBC_HW_ENC_PARAMS_RC_CONSTQP;
    encodeConfig.dwQP = 0;
}
```

Note: Lossless encoding will work only with constant QP rate control mode. If any other rate control mode is set, an error will be returned. Also, the quantization parameter value will be overridden to 0 and profile to 244.

2.9.10 Using YUV 4:4:4 Encoding with INVFBCToHWEncoder

YUV 4:4:4 encoding is useful in cases where chroma subsampling from RGB to YUV 4:2:0 will result in visible and unacceptable loss of video/image quality after encoding. Such loss is typically perceptible in regions with low luminance or blue/red text or wiremesh content (e.g. content with lines that are 1-2 pixels wide).

Client should call NVFBCHWEncGetCaps() to check for `NV_HW_ENC_GET_CAPS::bYUV444Supported` for the currently set codec type before configuring lossless encode, as not all NVIDIA GPUs support lossless encoding.

```
//! Example:

//! Check capability
NV_HW_ENC_GET_CAPS caps = {0};
caps.dwVersion = NV_HW_ENC_GET_CAPS_VER;
caps.eCodec = codec;
```

```
pNVFBCHWEnc->NVFBCHWEncGetCaps(&caps);

NV_HW_ENC_CONFIG_PARAMS encodeConfig = {0};

//! Other encode config init
if (caps.bYUV444Supported)
{
    encodeConfig.bEnableYUV444Encoding = 1;
}
```

Note: Some NVIDIA GPUs that are capable of HW encoding, are not capable of supporting YUV 4:44 video encoding. Client code should check HW capabilities as described above before enabling YUV4:4:4 encoding mode.

2.10 CAPTURING HW CURSOR ON SEPARATE THREAD

To enable grabbing the HW cursor separately, client must set the flag `bEnableSeparateCursorCapture` in the respective NVFBC interface setup parameters.

For example, for NVFBCToSys, set `NVFBC_TOSYS_SETUP_PARAMS::bEnableMouseGrab` while calling `NvFBCToSysSetup()`

Similarly, set `NVFBC_HW_ENC_SETUP_PARAMS::bEnableSeparateCurosrCapture` while calling `NvFBCHWEncSetup()`

If this flag is set, NVFBC will return a valid event handle in `hCursorCaptureEvent` member of the setup parameters. Eg: `NVFBC_TOSYS_SETUP_PARAMS::hCursorCaptureEvent`

Every time NVFBC captures an update to the cursor, this event will be signaled. The client should spawn a thread to wait on this event. The thread should wake up when the event is signalled and read back the cursor data.

Below is a sample code snippet to grab cursor glyph using NVFBCToSys:

Please refer to `NVFBC_CURSOR_CAPTURE_PARAMS` in NVFBC API Reference document for details.

```
//! Example Code
NVFBC_TOSYS_SETUP_PARAMS fbcSysSetupParams = {0};
//! Common configuration code here
//...
//! Set up separate HW cursor grab
fbcSysSetupParams.bEnableMouseGrab = true;
status = NVFBCToSys->NVFBCToSysSetUp(&fbcSysSetupParams);
hMouseEventHandle = fbcSysSetupParams.hCursorCaptureEvent;

NVFBC_CURSOR_CAPTURE_PARAMS pCursorCaptureParams;
pMouseGrabParams.dwVersion = NVFBC_CURSOR_CAPTURE_PARAMS_VER;
while(1)
{
    WaitForSingleObject(handle ,INFINITE); //handle returened from
//NVFBCToSysSetUp call
    ToSys->NVFBCToSysCursorCapture(&pCursorCaptureParams);
    if(pMouseGrabParams.bIsHwCursor)
    {
        out = base + _itoa(FrameID, frameNo, 10) + ".bmp";;
        SaveARGB(out.c_str(), (BYTE*) pCursorCaptureParams.pBits,
pCursorCaptureParams.dwWidth, pCursorCaptureParams.dwHeight);
        temp++;
    }

}
```

2.11 DIFFERENCE MAPS

Difference maps are supported for NVFBCToSys and NVFBCToDX9Vid interfaces.

The difference map format is a byte array, where each byte represents a block of the pixel region on the screen (in row-major order). If the byte is non-zero then some pixels have changed in that region. For resolutions that aren't a multiple of 128 in either direction, the resolution is rounded up to the next multiple of 128.

In order to get an accurate reconstruction of the grabbed images, the client application must apply each difference map since the last reference image. Discarding one or more difference maps may lead to corruption in reconstructed images. If the client application needs to discard a grabbed image, it should still merge the difference map with the previously captured difference map. Since difference maps are bit-arrays, the cost of condensing multiple difference maps into one buffer is very low: it can be achieved by OR-ing two 128-bit arrays.

2.11.1 Configuring Difference Map

NVFBC Capture SDK 6.1 adds support for generating difference maps with block size of 16x16, 32x32, 64x64 apart from legacy 128x128 blocksize. The client should call NvFBC_GetStatusEx() to determine if the driver supports configurable difference map block size. If `NvFBCStatusEx::bSupportConfigurableDiffMap` is set to 1 the driver supports configurable difference map blocksizes.

Any supported block size can be requested by setting the parameter `NVFBC_TODX9VID_SETUP_PARAMS::eDiffMapBlockSize` with one of the enum in `NVFBC_DX9VID_DIFFMAP_BLOCKSIZE`. This enum must be typecasted to `NvU32` before assigning. Eg:

```
NVFBC_TODX9VID_SETUP_PARAMS NvFBCDX9SetupParams = { 0 };

. . . . .

NvFBCDX9SetupParams.bDiffMap = TRUE;

NvFBCDX9SetupParams.eDiffMapBlockSize =

                          (NvU32)NVFBC_DX9VID_DIFFMAP_BLOCKSIZE_16X16;

NvFBCDX9SetupParams.dwDiffMapBuffSize = DIFF_MAP_BUF_SIZE;

NvFBCDX9SetupParams.ppDiffMap = (void **)&g_pDiffMap;

result = NvFBCDX9->NvFBCToDx9VidSetUp(&NvFBCDX9SetupParams));

. . . . .
```

Note: The client must call `NVFBC_GetStatusEx()` to check for configuring diference map block size. If `bSupportConfigurableDiffMap` is set to 1, then 16x16, 32x32, 64x64 and 128x128 difference map is supported. If client requests any of these blocksizes in setup parameters for an unsupported driver, NvFBC will ignore the request and return 128x128 difference map.

2.12 10 BIT AND HDR CAPTURE SUPPORT

2.12.1 NVFBC 10 bit capture support

NVIDIA Capture SDK 6.0 adds support for ARGB10 output format to the following interfaces: NvFBCToSys, NvFBCToDx9Vid and NvFBCToCuda.

To request 10 bit Capture, the client should use the 10 bit capture format enum from the corresponding interface. Eg: For NVFBCToSys, the client should use `NVFBC_TOSYS_ARGB10`.

Any 8 bit to 10 bit format conversion required will be taken care by the driver.

```
    NVFBC_TOSYS_SETUP_PARAMS fbcSysSetupParams = {0};

    fbcSysSetupParams.eMode = NVFBC_TOSYS_ARGB10;

    . . . . .

    result = nvfbcToSys->NvFBCToSysSetUp(&fbcSysSetupParams);

    . . . . .
```

2.12.2 NVFBC 10 bit HDR capture support

NVIDIA Capture SDK 6.0 adds support for capturing in HDR format if the display is configured for HDR, and the content is rendered in HDR.

To request HDR capture, the client should set the "`bHDRRequest`" flag in NVFBC setup parameters. A new flag, `NvFBCFrameGrabInfo::bIsHDR`, should be used to check if the current captured frame is in HDR. `NvFBCFrameGrabInfo::bIsHDR` will be set to 1 only if client requests HDR capture and NVFBC is able to capture in HDR format.

```
    NVFBC_TOSYS_SETUP_PARAMS fbcSysSetupParams = {0};
    fbcSysSetupParams.eMode = NVFBC_TOSYS_ARGB10;

    fbcSysSetupParams.bHDRRequest = TRUE;

    . . . . .

    result = nvfbcToSys->NvFBCToSysSetUp(&fbcSysSetupParams);

    . . . . .

    fbcSysGrabParams.pNvFBCFrameGrabInfo = &grabInfo;

    if (grabInfo.bIsHDR)

    {

          // captured content is HDR

    }
```

Note: 10 bit HDR HEVC encode support is available only for Pascal GPUs, using NVENC API. NVFBCToHWEnc interface does not have support for 10bit or HDR capture.

2.13 FACTORS REQUIRING NVFBC OBJECT RE-CREATION

Under the following circumstances, NVFBC objects must be destroyed and re-created in order to continue grabbing frames. This is indicated by `NVFBCFrameGrabInfo::MustRecreate` set to `TRUE` in the `NVFBCFrameGrabInfo` structure or `NVFBC_ERROR_SESSION_INVALIDATED` error code returned by a Grab call.

- The system transitions through an S3 (sleep) or S4 (hibernate) power state – for example, on a notebook platform, the user closes and later re-opens the notebook.
- Display topology is changed during an active NVFBC capture session.

Display capabilities are changed in a manner that changes the maximum display resolution, while an NVFBC capture session is running. Threading considerations

NVFBC is designed for use in multi-threaded applications, to allow for parallelism in handling readback operations on different heads, and in CPU-based post-processing of pixel data returned from NVFBC.

However, the API requires that all NVFBC API calls made on a given NVFBC object should be made from the thread that created the object. If an application uses an NVFBC object in different threads, each of those threads may have NVFBC API calls outstanding at the same time. Exceptions to this are as follows:

- In case a thread running an NVFBC session exits abnormally, the application should reset the directx9 device that was being used for NVFBC interaction, as the abnormal exit may leave the directx9 device context in an unstable state.
- The requirement is relaxed in case the client is using separate HW mouse cursor capture API in conjunction with NVFBCToSys. In this case, it is permitted to use NVFBCToSysCaptureMouse() and NVFBCToSysGrabFrame() on separate threads.

2.14 HANDLING ERRORS FROM NVFBC GRAB API

When NVFBC Grab succeeds, the API returns a status code `NVFBC_SUCCESS`. In case of failure, the API can return status codes depending upon the error. The status codes are documented in NVFBC API reference. In addition, `NvFBCFrameGrabInfo::dwDriverInternalError` holds more information about the failure. This is intended to be used as diagnostic information while investigating failures. If the API succeeds, this code should be ignored.

The table below outlines some values that can be actionable for the client application:

Value	**Meaning**
`0xFBCB0001`	Non-fatal. Grab was called while a resolution change or display topology change was in progress
`0xFBCB0002`	Non-fatal. Invalid grab flags, retry with a valid value for grab flags. Current request was treated as if no flags were set.
`0xFBCB0003`	Non-fatal. No screen update for 1 second after the capture session was created, output is black.
`0xFBCE0003`	Non-fatal. Invalid parameters, retry with correct parameters.
`0xFBCE0004`	Fatal. Invalid sequence of API calls or Capture session is in invalid state. Release capture session and create new session.
`0xFBCE0027`	Non-fatal. Invalid cropping rect, retry with valid cropping rect based on current and maximum display dimensions.
`0xFBCE0028`	Non-fatal. Invalid scaling target rect, retry with valid target rect.
`0xFBCE0044`	Fatal. NVFBC feature was disabled since the last Grab(). Release capture session.

Table 3: NVFBC Grab API Diagnostic codes

2.14.1 Handling protected content

Playback of protected content such as DVD or BluRay disks typically requires an encrypted, secured path to the physical display output device. To prevent violation of protected content license terms, NVFBC will not capture frames from the GPU whenever a protected content session is active.

NVFBC indicates the presence of protected content by returning `NVFBC_ERROR_PROTECTED_CONTENT` from calls to `NVFBC GrabFrame()`, and no frame data is returned.

While protected content is active, applications should fall back to non-accelerated, standard Windows APIs to capture the desktop without the protected content, in order to continue providing visual output to the user. In parallel with this, NVFBC API frame grabs should be attempted periodically to determine when the protected content session has ended, and accelerated frame grabbing is once again possible. Figure 4 summarizes the suggested program flow:

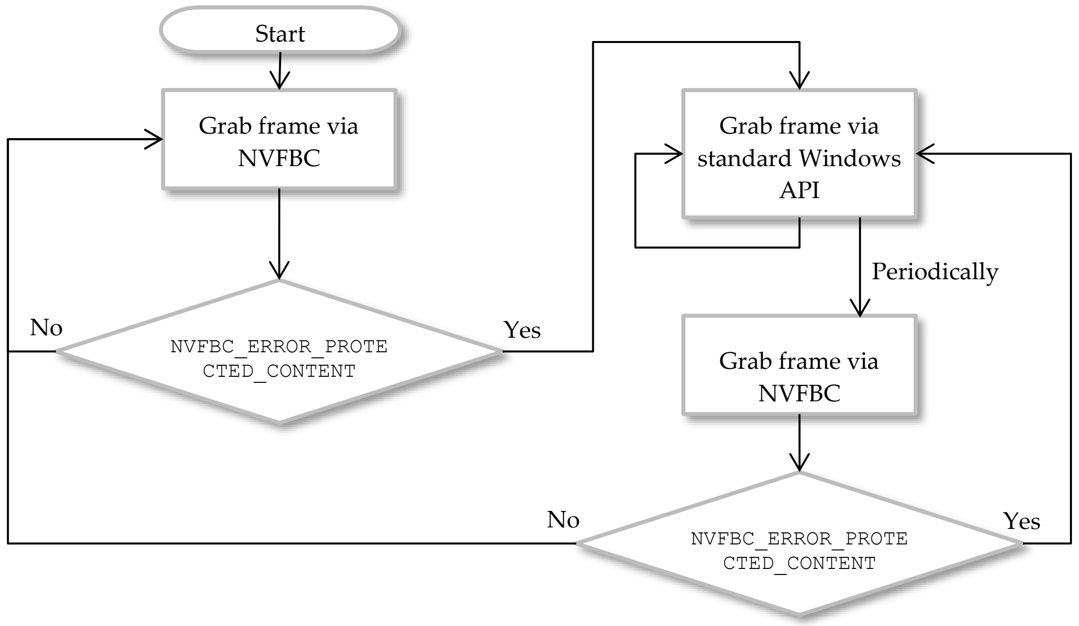

Start
Grab frame via NVFBC
NVFBC_ERROR_PROTECTED_CONTENT
No
Yes
Grab frame via standard Windows API
Periodically
Grab frame via NVFBC
NVFBC_ERROR_PROTECTED_CONTENT
No
Yes

Figure 4 Handling protected content

2.14.2 Handling an Invalidated Session

If the error code NVFBC_ERROR_INVALIDATED_SESSION is returned when attempting a frame grab, the user must re-create the NVFBC session by:

- Destroying allocated buffers and closing event handles.
- Releasing NVFBC by calling the Release() API of the corresponding interface.
- If possible, release the DX9 device that was passed to NVFBC, and create a new device.
- Following the NVFBC initialization steps again.

```
pNVFBCHWEnc->NVFBCHWEncRelease();

pNVFBCHWEnc= (NVFBCToHWEncoder*) pfnNVFBC_Create (NVFBC_TO_
HW_ENCODER, &maxWidth, &maxHeight);

//! Setup the grab and encode

NVFBCRESULT result;
…
result = pNVFBCHWEnc->NVFBCHWEncSetUp(&setupParams);
```

Chapter 3. NVIFR – INBAND FRAME READBACK

NVIDIA Inband Frame Readback (NVIFR) is a high performance, low latency API for capturing and optionally compressing an individual DirectX graphics render target. The output from NVIFR does not include any window manager decoration, composited overlay, cursor or taskbar; it provides only the pixels rendered into the render target, as soon as their rendering is complete, ahead of any compositing that may be done by the windows manager. In fact, NVIFR does not require that the render target even be visible on the Windows desktop. It is ideally suited to *application capture and remoting*, where the output of a single application, rather than the entire desktop environment, is captured.

NVIFR is intended to operate *inband* with a rendering application, either as part of the application itself, or as part of a shim layer operating immediately below the application. Like NVFBC, NVIFR operates asynchronously to graphics rendering, using dedicated hardware compression and copy engines in the GPU, and delivering pixel data to system memory with minimal impact on rendering performance.

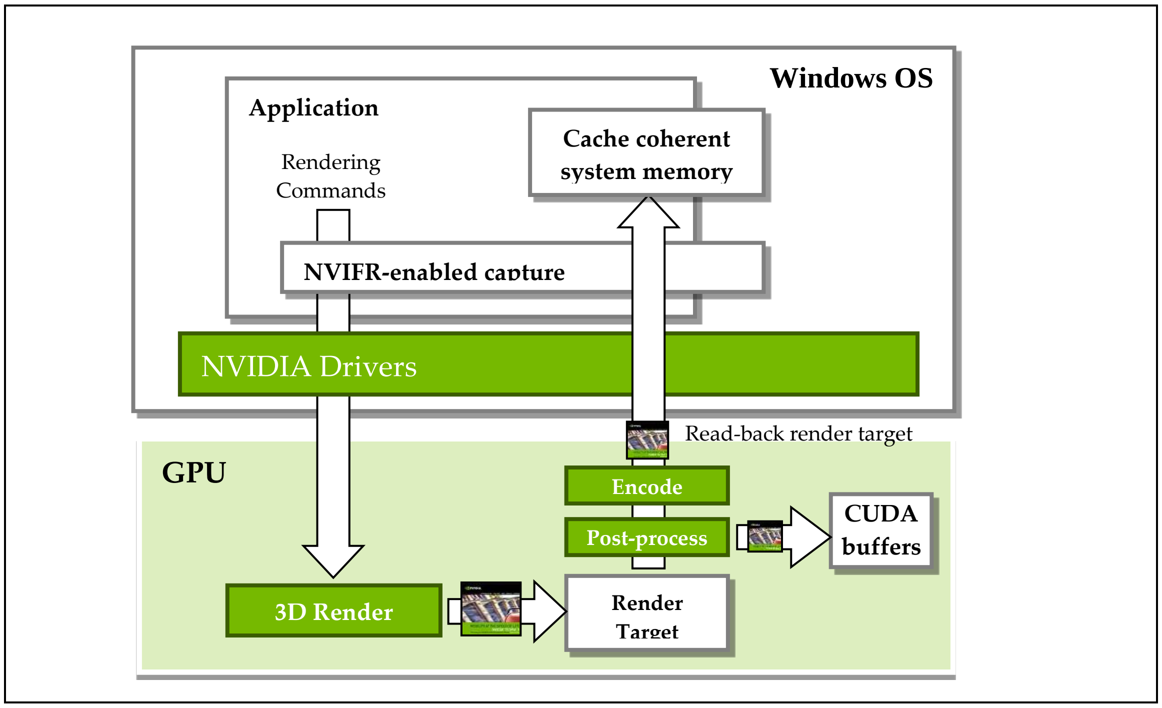


Figure 5 NVIFR render context capture

3.1 HEADER FILES AND CODE SAMPLES

This manual provides an overview of how to use NVIFR. Further details are contained in the NVIFR header files and code samples that are included in the NVIDIA Capture SDK Toolkit:

The NVIFR header files, including interface-specific is installed in `%CAPTURESDK_PATH%\inc\NVIFR\`. All NVIFR applications don't need to include `NVIFR.h` directly, as the specific versions of the NVIFR interfaces include it .

NVIFR code samples are installed in `%CAPTURESDK_PATH%\samples\`

Please refer to the NVIDIA Capture SDK Samples Description document for details regarding the NVIFR samples packaged in the NVIDIA Capture SDK.

3.2 PREPARING THE API FOR USE

Regardless of the mode in which an application uses the NVIFR API, the following initialization steps are required at application load time:

- Load the NVIFR DLL
- Loading the NVIFR_CreateEx function pointer

3.2.1 Loading the DLL

The NVIFR API is accessed via a 32- or 64- bit dynamic link library (DLL), which must be loaded by the application before calling any NVIFR functions:

```
// Load an instance of the NVIFR DLL

// 32-bit application
HINSTANCE handleNVIFR = ::LoadLibrary("NVIFR.dll");

// 64-bit application
HINSTANCE handleNVIFR = ::LoadLibrary("NVIFR64.dll");
```

Note: in the NVIDIA Capture SDK toolkit, the NVIFR DLLs are located in `…\<toolkit install dir>\lib\NVIFR`. When shipping an application that uses the NVIDIA Capture SDKs, we recommend the DLLs be installed in the same directory as your application. The pathname passed to `::LoadLibrary()` should be amended accordingly. See Chapter 4 for further guidance on shipping GRID-enabled applications.

3.2.2 Loading the function pointer

After loading the NVIFR DLL, the next step is to get the `NVIFR_Create()` function pointer from the DLL. This is accomplished with a call to `GetProcAddress()`:

```
// Load the NVIFR_Create funtion

NVIFR_CreateFunctionExType pfnNVIFR_Create = NULL;

pfnNVIFR_Create = (NVIFR_CreateFunctionExType)
     GetProcessAddress(handleNVIFR, "NVIFR_CreateEx");
```

3.3 CREATING NVIFR OBJECTS

3.3.1 Creating Objects

All NVIFR readback operations are exposed as methods in NVIFR classes. Distinct classes are used to support the different readback modes supported by NVIFR (readback to system memory and readback as compressed video).

To create an NVIFR object you must create an instance of `IDirect3D9Device` or `ID3D10Device` or `ID3D11Device`. This device interface is passed into `NVIFR_CreateEx()` along with the readback format:

```
NVIFRRESULT (__stdcall *NVIFR_CreateFunctionExType) (void *pParams);

// Example usage
ID3D10Device * pDevice = NULL;
D3D10CreateDeviceAndSwapChain(…, &pDevice); // Create the device
NVIFR_CREATE_PARAMS params = {0};
params.version = NVIFR_CREATE_PARAMS_VER;
params.dwInterfaceType = NVIFR_TOSYS;
params.pDevice = pDevice;

NVIFRToSys * toSys = NULL;
NVIFRRESULT res = pfnNVIFR_Create(&params);
if (res == NVIFR_SUCCESS)
{
    toSys = (NVIFRToSys *)params.pNVIFR;

}
```

If successful, the `NVIFR_Create()` call returns a pointer to the newly-created NVIFR object, otherwise it returns `NULL`.

The `dwInterfaceType` parameter specifies the type of NVIFR object to create:

`dwInterfaceType` value	**Notes**
`NVIFR_TOSYS`	Reads back frames to locked, cache-coherent buffers in system memory. See section 3.4.
`NVIFR_TO_HW_ENCODER`	Reads back compressed frames to locked, cache-coherent buffers in system memory. See section 3.5.

Table 4 NVIFR interface Types

The `pdwNVIFRVersion` parameter is an output argument which will contain the NVIFR version after the call has been completed.

3.3.2 Limitations

There are certain limitations when creating multiple NVIFR objects using the same DirectX device:

- Creating NVIFRToSys and NVIFRToHWEncoder objects using the same DirectX device may result in undefined behavior.
- If an application creates multiple NVIFRToHWEncoder objects using the same DirectX device, it should also take care to release all objects at one go. Releasing one object and continuing to use other objects will lead to undefined behavior. The same applies for NVIFRToSys.
- Creating multiple NVIFRToHWEncoder objects that produce output videos of different frame sizes is supported but is suboptimal for performance. The same applies for NVIFRToSys.

These are known limitations in the NVIDIA Capture SDK up to NVIDIA Capture SDK 6.0, and may be fixed in future revisions.

3.4 CAPTURING TO SYSTEM MEMORY

To capture render targets to system memory, create an NVIFRToSys object by specifying `NVIFR_TOSYS` in the `NVIFR_Create()` call:

```
// Create an instance of NVIFRToSys
NVIFRToSys *toSys = pfnNVIFR_Create(device, NVIFR_TOSYS, &version);
```

3.4.1 Setting up the target buffers

`SetupTargetBufferToSys()` must be called before reading back render target buffers.

```
NVIFRRESULT NVIFRSetUpTargetBufferToSys(NVIFR_TOSYS_SETUP_PARAMS
*pParams)

// Example usage
#define NUMFRAMESINFLIGHT = 3
unsigned char *buffer;
HANDLE gpuEvent;

NVIFR_TOSYS_SETUP_PARAMS params = {0};
params.dwVersion = NVIFR_TOSYS_SETUP_PARAMS_VER;
params.eFormat = NVIFR_FORMAT_RGB;
params.eSysStereoFormat = NVIFR_SYS_STEREO_NONE;
params.dwNBuffers = 1;
params.ppPageLockedSysmemBuffers = buffer;
params.ppTransferCompletionEvents =  &gpuEvent;

NVIFRRESULT result = toSys->NVIFRSetUpTargetBufferToSys(&params);
```

If succesful, `NVIFRSetUpTargetBufferToSys()` returns `NVIFR_SUCCESS` and the object is ready to transfer the render target. Otherwise it returns one of the error codes specified by `NVIFRRESULT` in `NVIFR.h`.

3.4.1.1 Capture mode

The `NVIFR_TOSYS_SETUP_PARAMS::eFormat` field specifies the pixel format in which the render target content is captured. Please refer to `NVIFR_BUFFER_FORMAT` enum in NVIFR API Reference document.

3.4.1.2 Stereo Format

The `NVIFR_TOSYS_SETUP_PARAMS::eSysStereoFormat` field specifies the stereo format to capture. Please refer to `NVIFR_SYS_STEREO_FORMAT` enum in NVIFR API Reference document.

This is currently not supported.

3.4.1.3 Number of output buffers

The `NVIFR_TOSYS_SETUP_PARAMS::dwNBuffers` field specifies the number of output buffers to generate. The `NVIFRToSys` interface allows you to use multiple output buffers. The number of output buffers generated is equal to `NVIFR_TOSYS_SETUP_PARAMS::dwNBuffers`.

3.4.1.4 Readback buffers

The `NVIFR_TOSYS_SETUP_PARAMS::ppPageLockedSysmemBuffers` field is an an array of pointers. Each pointer will point to a buffers that will contain the read back pixel data, in the format specified by the capture mode in `NVIFR_TOSYS_SETUP_PARAMS::eFormat`.

The buffers are allocated by NVIFR, the application simply passes a `unsigned char**` argument to receive a pointers to the NVIFR allocated buffers. The number of buffers generated depends on the `NVIFR_TOSYS_SETUP_PARAMS::dwNBuffers` parameter.

3.4.1.5 Completion events

The `NVIFR_TOSYS_SETUP_PARAMS::ppTransferCompletionEvents` field is a pointer to an array of Event `HANDLEs` that will be created by NVIFR. There number of `HANDLEs` must be equal to the `NVIFR_TOSYS_SETUP_PARAMS::dwNBuffers` parameter.

3.4.1.6 Scaling

The `NVIFR_TOSYS_SETUP_PARAMS::dwTargetWidth` and `NVIFR_TOSYS_SETUP_PARAMS::dwTargetHeight` fields are optional, and may be specified to apply scaling to the captured render target before transferring to the output buffer. If either parameter is set to 0 then no scaling is performed. Otherwise the width and height of the output image are set to the specified values.

3.4.2 Transferring the render target

Transfering the render target is an asynchronous operation. Call `NVIFRTransferRenderTargetToSys()` to initiate a transfer; when the transfer is complete, an Event `HANDLE` that was earlier provided to `NVIFRSetUpTargetBufferToSys()` will be signaled.

```
NVIFRRESULT NVIFRTransferRenderTargetToSys(NvU32 dwBufferIndex));

//! Example usage

NVIFRRESULT result;
result = toSys->NVIFRTransferRenderTargetToSys(0);

WaitForSingleObject(gpuEvent);

//! Render target is now transferred into system memory
```

3.4.2.1 Buffer Index

The `dwBufferIndex` parameter is used to specify which locked system memory buffer and completion event `HANDLE` will be used for the transfer. The `dwBufferIndex` must be less than the `NVIFR_TOSYS_SETUP_PARAMS::dwNBuffers` parameter passed to the `SetUpTargetBufferToSys()` call.

For multi-threading:
-In main thread, Call `NVIFRTransferRenderTargetToSys` with `dwBufferIndex` 'i' in a loop where `0 < i < NVIFR_TOSYS_SETUP_PARAMS::dwNBuffers`
- In a worker thread, in a loop, call `WaitForSingleObject(gpuEvent[i]);` where `0 < i < dwNumBuffers`
- Consume grabbed image from the worker thread.

3.4.3 Releasing the NVIFRToSys object

After you have finished using `NVIFRToSys` you must release it to properly free the resources.

```
NVIFRRESULT NVIFRRelease();

//! Example code
toSys->NVIFRRelease();
```

3.5 CAPTURING WITH HARDWARE VIDEO COMPRESSION

To capture compressed video frames to system memory, create an `INvIFRHWEncoder` object by specifying `NVIFR_TO_HWENCODER` in the `NVIFR_CreateEx()` call:

```
//! Create an instance of INVIFRHWEncoder
IDirect3DDevice9 * pDevice = NULL;
IDirect3D9 *pD3D = NULL;

InitD3D(pD3D, pDevice); // Create the device

NVIFR_CREATE_PARAMS params = {0};
params.version = NVIFR_CREATE_PARAMS_VER;
params.dwInterfaceType = NVIFR_ TO_HWENCODER;
params.pDevice = pDevice;

INVIFRHWEncoder * toHWEnc = NULL;
NVIFRRESULT res = pfnNVIFR_CreateEx(&params);
if (res == NVIFR_SUCCESS)
{
    toHWEnc = (INVIFRHWEncoder *)params.pNVIFR;

}
```

Note: Not all NVIDIA GPUs that support NVIFR support hardware video compression. If the GPU does not support hardware video compression, the `NVIFR_CreateEx()` call for an `NVIFR_TO_HWENCODER` object will return `NVIFR_CREATE_PARAMS::pNVIFR==NULL`.

3.5.1 Checking HW Video Encoder Capabilities

`INVIFRHWEncoder::NVIFRGetHWEncCaps()` can be used to check HW Video encoder capabilities like supported codec, rate control modes, encoding presets, etc. Please refer to the NVIFR API Reference document for details.

This should be done before calling `INVIFRHWEncoder:: NVIFRSetUpHWEncoder()`.

3.5.2 Setting up the target buffers

Before the compressed buffers can be copied into system memory they need to be set up. This is accomplished by calling `NVIFRSetUpHWEncoder()`.

```
//! Example usage
#define N_BUFFERS 3

//! container for pagelocked sysmem buffers for encoded bitstream
unsigned char * g_pBitStreamBuffer[N_BUFFERS];
//! event handles used to signal encode completion
HANDLE g_EncodeCompletionEvent[N_BUFFERS];
NVIFR_HW_ENC_SETUP_PARAMS params = {0};
params.dwVersion = NVIFR_HW_ENC_SETUP_PARAMS_VER;

//! eCodec must be NV_HW_ENC_H264 or NV_HW_ENC_HEVC
params.configParams.eCodec = eCodec;
params.dwNBuffers = N_BUFFERS;
params.dwBSMaxSize = 2048*1024;
params.ppPageLockedBitStreamBuffers = &g_pBitStreamBuffer;
params.ppEncodeCompletionEvents = &g_EncodeCompletionEvent;

NV_HW_ENC_CONFIG_PARAMS encodeConfig = {0};
encodeConfig.dwVersion = NV_HW_ENC_CONFIG_PARAMS_VER;
encodeConfig.dwProfile = 100;
encodeConfig.dwAvgBitRate = (DWORD)dBitRate;
encodeConfig.dwFrameRateDen = 1;
encodeConfig.dwFrameRateNum = 30;
encodeConfig.dwPeakBitRate =
            (encodeConfig.dwAvgBitRate * 12/10);   // +20%
encodeConfig.dwGOPLength = 75;
encodeConfig.dwQP = 26 ;
encodeConfig.eRateControl = NV_HW_ENC_PARAMS_RC_CBR;
encodeConfig.ePresetConfig= NV_HW_ENC_PRESET_LOW_LATENCY_HQ;

//! Set Encode Config
params.configParams = encodeConfig;

NVIFRRESULT res = toHWEnc->NVIFRSetUpHWEncoder(&params);
```

If succesful, `NVIFRSetUpHWEncoder()` returns `NVIFR_SUCCESS` and the `NVIFRToHWEncoder` object is ready to transfer the render target. Otherwise it returns one of the error codes specified by `NVIFRRESULT` in `NVIFR.h`.

3.5.2.1 HW encoding parameters

The `NV_HW_ENC_CONFIG_PARAMS` structure defines encoding parameters to be used by the hardware encoder. Please refer to NVIFR API Reference document for details. . PL

3.5.2.2 Number of output buffers

The `NVIFR_HW_ENC_SETUP_PARAMS::dwNBuffers` field specifies the number of output buffers to generate. The `INVIFRHWEncoder` interface allows you to use multiple output buffers. The number of output buffers generated is equal to `dwNBuffers` reguardless of the stereo mode.

3.5.2.3 Maximum bitstream size

The `NVIFR_HW_ENC_SETUP_PARAMS::dwBSMaxSize` field sets the maximum size of a bitstream frame.

3.5.2.4 Output buffers

The `NVIFR_HW_ENC_SETUP_PARAMS::ppPageLockedBitStreamBuffers` field is a pointer to pointer to a buffer that will contain the read back frame data, in the format specified by the capture mode in `eFormat`. The buffers are allocated by NVIFR, the application simply passes a `unsigned **` argument to receive a pointers to the NVIFR allocated buffers. The number of buffers generated depends on the `dwNBuffers` parameter.

3.5.2.5 Completion events

The `NVIFR_HW_ENC_SETUP_PARAMS::ppEncodeCompletionEvents` field is a pointer to an array of `HANDLEs` created by the application. There number of `HANDLEs` must be equal to the `dwNBuffers` parameter.

3.5.2.6 Scaling

The `NVIFR_HW_ENC_SETUP_PARAMS::dwTargetWidth` and `NVIFR_HW_ENC_SETUP_PARAMS::dwTargetHeight` fields are optional, and may be specified to apply scaling to the captured render target before encode. If either parameter is set to 0 then no scaling is performed. Otherwise the width and height of the output video stream are set to the specified values.

3.5.3 Transferring the render target

Transferring the compressed render target is an asynchronous operation. Call `TransferRenderTargetToHWEncoder()` to initiate a transfer. When the transfer is complete the corresponding `NVIFR_HW_ENC_SETUP_PARAMS::ppEncodeCompletionEvent` is signaled. The size of the compressed frame written to the output buffer can be queried by calling `NVIFRGetStatsFromHWEncoder()`.

```
NVIFRTransferRenderTargetToHWEncoder
        (NVIFR_HW_ENC_TRANSFER_RT_TO_HW_ENC_PARAMS *pParams);

//! Example usage
NV_HW_ENC_PIC_PARAMS encodePicParams = {0};
encodePicParams.dwVersion = NV_HW_ENC_PIC_PARAMS_VER;

NVIFR_HW_ENC_TRANSFER_RT_TO_HW_ENC_PARAMS params = {0};
params.dwVersion = NVIFR_HW_ENC_TRANSFER_RT_TO_HW_ENC_PARAMS_VER;
params.dwBufferIndex = i;
params.encodePicParams = encodePicParams;

NVIFRRESULT res =
    toHWEnc->NVIFRTransferRenderTargetToHWEncoder(&params);

if (res == NVIFR_SUCCESS) {
    WaitForSingleObject(g_EncodeCompletionEvents[i], INFINITE);
}
```

3.5.3.1 Buffer Index

The `NVIFR_HW_ENC_TRANSFER_RT_TO_HW_ENC_PARAMS::dwBufferIndex` field is used to specify which locked system memory buffer and completion event object will be used for the transfer. The `NVIFR_HW_ENC_TRANSFER_RT_TO_HW_ENC_PARAMS::dwBufferIndex` must be less than the `NVIFR_HW_ENC_SETUP_PARAMS::dwNBuffers` parameter passed into the `NVIFRSetUpHWEncoder()` call.

3.5.3.2 Per frame encoding parameters

The `NVIFR_HW_ENC_TRANSFER_RT_TO_HW_ENC_PARAMS::encodePicParams` parameter is used to specify some per-frame customizations for the HW Encoder. Please refer to `NV_HW_ENC_PIC_PARAMS` in the NVIFR API Reference document for details.

3.5.4 Getting the frame stats

`NVIFRGetStatsFromHWEncoder()` should be called on receiving the encode completion event signal, to get the frame stats for the encoded frame.

```
NVIFRRESULT NVIFRGetStatsFromHWEncoder
              NVIFR_HW_ENC_GET_BITSTREAM_PARAMS *pParams);

//! Example usage
NVIFR_HW_ENC_TRANSFER_RT_TO_HW_ENC_PARAMS params = {0};
params.dwVersion = NVIFR_HW_ENC_TRANSFER_RT_TO_HW_ENC_PARAMS_VER;
params.dwBufferIndex = i;
params.encodePicParams = encodePicParams;

NV_HW_ENC_PIC_PARAMS encodePicParams = {0};
encodePicParams.dwVersion = NV_HW_ENC_PIC_PARAMS_VER;

NVIFRRESULT res =
    toHWEnc->NVIFRTransferRenderTargetToHWEncoder(&params);

if (res == NVIFR_SUCCESS)
{
    WaitForSingleObject(g_EncodeCompletionEvents[i], INFINITE);
    ResetEvent(g_EncodeCompletionEvent[i]);

    //! Get frame stats from the HWencoder
    NVIFR_HW_ENC_GET_BITSTREAM_PARAMS params = {0};
    params.dwVersion = NVIFR_HW_ENC_GET_BITSTREAM_PARAMS_VER;
    params.bitStreamParams.dwVersion =
                         NV_HW_ENC_GET_BIT_STREAM_PARAMS_VER;
    params.dwBufferIndex = i;
    NVIFRRESULT res = toHWEnc->NVIFRGetStatsFromHWEncoder(&params);

    //! Can Write new data to disk

}
```

If successful, `NVIFRGetStatsFromHWEncoder()` returns `NVIFR_SUCCESS`, and the size of the bitstream in bytes is written to `NV_HW_ENC_GET_BIT_STREAM_PARAMS::dwByteSize` passed as `NVIFR_HW_ENC_GET_BITSTREAM_PARAMS::bitStreamParams`. Otherwise it returns one of the error codes specified by `NVIFRRESULT` in `NVIFR.h`.

3.5.4.1 Buffer Index

The `NVIFR_HW_ENC_GET_BITSTREAM_PARAMS::dwBufferIndex` field is used to specify the output buffer to read, and must be less than the `NVIFR_HW_ENC_SETUP_PARAMS::dwNBuffers` parameter passed into the `NVIFRSetUpHWEncoder()` call.

3.5.4.2 Frame Stats

The `NVIFR_HW_ENC_GET_BITSTREAM_PARAMS::bitStreamParams` parameter returns the stats for the encoded frame. Refer to NVIFR API Reference document for details regarding `NV_HW_ENC_GET_BIT_STREAM_PARAMS`.

3.5.5 Reading Sequence and Picture Parameter Sets

Once the `NVIFRToHWEncoder` setup is successfully completed, a client can call `NVIFRGetHeaderFromHWEncoder`() API at any time during the session to fetch the current H.264 or H.265 Sequence parameter set and picture parameter set.

The API will populate a client allocated buffer with SPS NALU followed by PPS NALU.

```
NVIFRRESULT
NVIFRGetHeaderFromHWEncoder(NVIFR_HW_ENC_GET_STREAM_HEADER_PARAMS
*pParams)

//! Example Usage
#define MAX_SPS_PPS_HEADER_SIZE 1024*1024;
unsigned char buffer[MAX_SPS_PPS_HEADER_SIZE];
int dwSize = 0;
NVIFR_HW_ENC_GET_STREAM_HEADER_PARAMS headerParms = {0};

Memset(buffer, 0, MAX_SPS_PPS_HEADER_SIZE);
headerParams.dwVersion = NVIFR_HW_ENC_GET_STREAM_HEADER_PARAMS_VER;
headerParams.pBuffer   = &buffer;
headerParams.pSize     = &dwSize;

NVIFRRESULT result = NVIFR_SUCCESS;
result = toHWEnc->NVIFRGetHeaderFromHWEncoder(&headerParms);
```

3.5.6 Releasing the INVIFRHWEncoder object

After you have finished using `NVIFRToHWEncoder` you must release it to properly free the resources.

```
NVIFRRESULT Release();

//! Example Usage
toHWEnc->Release();
```

3.5.7 Using Intra-Refresh with INVIFRHWEncoder

Intra-Refresh is an error-resiliency feature supported by NVIDIA Capture SDK, which allows the client to enable gradual decoder refresh or intra-refresh. This is supported only for GOP structures that do not use B-frames.

For example:

> If Intra-Refresh cycle count = n and no. of Macroblocks per frame = m,
>
> Then for 1st frame, 0 to (m/n) - 1 macroblocks are coded as intra-predicted, for 2nd frame (m/n) to (2m/n) – 1 macroblocks are coded as intra-predicted, and so on.

To use this feature, the client should set

```
NV_HW_ENC_CONFIG_PARAMS::bEnableIntraRefresh = 1;
```

before calling `NVIFRSetUpHWEncoder()`.

To indicate start of an Intra-refresh cycle, the client should set

```
NV_HW_ENC_PIC_PARAMS::bForceIntraRefresh = 1;
NV_HW_ENC_PIC_PARAMS::dwIntraRefreshCnt = n;
```

before calling `NVIFRTransferRenderTargetToHWEncoder()`.

At this point, NVIFR will begin an Intra-Refresh cycle, spread out over ‘n’ frames.

```
NVIFRTransferRenderTargetToHWEncoder
        (NVIFR_HW_ENC_TRANSFER_RT_TO_HW_ENC_PARAMS *pParams);

//! Example usage
NVIFR_HW_ENC_TRANSFER_RT_TO_HW_ENC_PARAMS params = {0};
params.dwVersion = NVIFR_HW_ENC_TRANSFER_RT_TO_HW_ENC_PARAMS_VER;
params.dwBufferIndex = i;

//! Start an Intra-Refresh cycle over n frames.
NV_HW_ENC_PIC_PARAMS encodePicParams = {0};
encodePicParams.dwVersion = NV_HW_ENC_PIC_PARAMS_VER;
encodePicParams.bForceIntraRefresh = 1;
encodePicParams.dwIntraRefreshCount = n;
params.encodePicParams = encodePicParams;

NVIFRRESULT res =
    toHWEnc->NVIFRTransferRenderTargetToHWEncoder(&params);

if (res == NVIFR_SUCCESS)
{
    WaitForSingleObject(g_EncodeCompletionEvents[i], INFINITE);

}
```

Note: Using Reference frame Invalidation in conjunction with Intra-Refresh is not supported. If client has enabled Intra-Refresh while setting up INVIFRHWEncoder,ference frame invalidation requests will be ignored.

Also, Intra-refresh takes higher precedence over Dynamic Slice Mode [Ref. Section

3.5.8] settings. Intra-refresh will always divide the picture in an integral number of slices depending upon the value specified for `NV_HW_ENC_PIC_PARAMS::dwIntraRefreshCnt`.

3.5.8 Using Dynamic Slice mode with INVIFRHWEncoder

Dynamic Slice mode encoding allows the client to configure how the encoded picture will be divided into slices. This is done by setting two parameters: `NV_HW_ENC_CONFIG_PARAMS::dwSlicingMode` and `NV_HW_ENC_CONFIG_PARAMS::dwSlicingModeParam` before calling `NVIFRSetUpHWEncoder()` .

Their usage is summarized in the NVIFR API Reference document.

Note: Intra-refresh takes higher precedence over Dynamic Slice Mode [Ref. Section 3.5.8] settings. Intra-refresh will always divide the picture in an integral number of slices depending upon the value specified for `NV_HW_ENC_PIC_PARAMS::dwIntraRefreshCnt`.

3.5.9 INVIFRHWEncoder Rate Control Modes

The rate control modes supported by INVIFRHWEncoder are described in the NVIFR API Reference document.

3.5.10 Using Adaptive Quantization with INVIFRToHWEncoder

Adaptive Quantization (AQ) can be controlled using `NV_HW_ENC_CONFIG_PARAMS::bEnableAdaptiveQuantization` flag.

Quantization artifacts like blockiness in a flat region are more visible than in a complex region. With AQ enabled, the goal is to improve quality in a flat region. With Adaptive Quantization enabled, the quantization parameter is set depending upon complexity of macroblock data, thus assigning higher values of QP for macroblocks in high complex regions and lower values of QP for macroblocks in flat regions, thereby improving the visual quality of flat regions.

Note: Adaptive quantization works only with 2 pass rate control modes

3.5.11 Using Lossless encoding with INVIFRHWEncoder

To retain the same quality after encoding-decoding, a lossless encoding feature is introduced. There is no loss of data in this mode; however, the size of the bit stream is large compared to lossy encoding.

Client should call `NVIFRGetHWEncCaps()` to check for `NV_HW_ENC_GET_CAPS::bLosslessEncodingSupported` for the currently set codec type before configuring lossless encode, as not all NVIDIA GPUs support lossless encoding.

```
//! Example:
//! Check capability
NV_HW_ENC_GET_CAPS caps = {0};
caps.dwVersion = NV_HW_ENC_GET_CAPS_VER;
caps.eCodec = codec;
toHWEnc-> NVIFRGetHWEncCaps (&caps);

NV_HW_ENC_CONFIG_PARAMS encodeConfig = {0};
//! Other encode config init
if (caps.bLosslessEncodingSupported)
{
    encodeConfig.dwProfile = 244;
    encodeConfig.ePresetConfig= NVFBC_HW_ENC_PRESET_LOSSLESS_HP;
    encodeConfig.eRateControl = NVFBC_HW_ENC_PARAMS_RC_CONSTQP;
    encodeConfig.dwQP = 0;
}
```

Note: Lossless encoding will work only with constant QP rate control mode. If any other rate control mode is set, an error will be returned. Also, the quantization parameter value will be overridden to 0 and profile to 244.

3.5.12 Using YUV 4:4:4 Encoding with INVIFRHWEncoder

YUV 4:4:4 encoding is useful in cases where chroma subsampling from RGB to YUV 4:2:0 will result in visible and unacceptable loss of video/image quality after encoding. Such loss is typically perceptible in regions with low luminance or blue/red text or wiremesh content (e.g. content with lines that are 1-2 pixels wide).

Client should call NVFBCHWEncGetCaps() to check for `NV_HW_ENC_GET_CAPS::bYUV444Supported` for the currently set codec type before configuring lossless encode, as not all NVIDIA GPUs support lossless encoding.

```
//! Example:

//! Check capability
NV_HW_ENC_GET_CAPS caps = {0};
caps.dwVersion = NV_HW_ENC_GET_CAPS_VER;
caps.eCodec = codec;
pNVFBCHWEnc->NVFBCHWEncGetCaps(&caps);

NV_HW_ENC_CONFIG_PARAMS encodeConfig = {0};

//! Other encode config init
if (caps.bYUV444Supported)
{
    encodeConfig.bEnableYUV444Encoding = 1;
}
```

Note: Some NVIDIA GPUs that are capable of HW encoding, are not capable of supporting YUV 4:44 video encoding. Client code should check HW capabilities as described above before enabling YUV4:4:4 encoding mode.

Chapter 4. DEPLOYING A GRID-ENABLED APPLICATION

This chapter provides guidance on shipping a GRID-enabled application.

4.1 DEPLOYMENT ON WINDOWS

Deploying an NVFBC- and NVIFR-enabled application with GeForce, Quadro, and Tesla GPUs on Windows platforms requires that you deploy a Microsoft DirectX redistributable runtime along with your application, and execute an install-time applet to set NVIDIA Capture SDK's required registry settings.

4.1.1 Microsoft DirectX redistributable runtime

The NVIDIA Capture SDK DLLs are linked with version 33 of the Microsoft DirectX runtime. As this version may not be present on an end user's platform, you should include the Microsoft redistributable end-user runtime with your application. The redistributable can be downloaded from here:

http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=8109

4.1.2 DLL installation

When shipping an NVFBC or NVIFR-enabled application, you do not need to include the NVFBC and/or NVIFR DLLs with your shipping application, as they are included in NVIDIA driver releases after 320.00.

DLL	Install Path	Notes
`NVFBC.DLL`	For 32Bit OS: %systemroot%\system32 For 64Bit OS: %systemroot%\syswow64	Application should load the correct version [32bit\64bit] of the DLLs based on the OS and application's target architecture from the paths listed here. Application should not package these DLLs with their installers, as they will be installed along with NVIDIA drivers.
`NVFBC64.DLL`	For 64Bit OS only. %systemroot%\system32	
`NVIFR.DLL`	For 32Bit OS: %systemroot%\system32 For 64Bit OS: %systemroot%\syswow64	
`NVIFR64.DLL`	For 64Bit OS only. %systemroot%\system32	

Table 5 NVIDIA Capture SDK DLL Path Names, Install Locations

The DLL locations in the NVIDIA Capture SDK Toolkit are shown in Table 24 above.

4.1.3 Registry settings

The NVIDIA Capture SDK's NVFBC component requires some registry settings to be present on the system to operate correctly. Enable and disable of these registry settings is abstracted into a simple executable, `NVFBCEnable.exe`, which should be shipped with your application install package and executed during application installation.

- To enable NVFBC registry settings, execute: `NVFBCEnable -enable`.
- To disable NVFBC registry settings, execute: `NVFBCEnable -disable`.
- To check status of NVFBC on your system, execute: `NVFBCEnable -checkstatus`

`NVFBCEnable` is provided in …`\<toolkit install dir>\bin` in the GRID toolkit. After changing the registry settings, this tool will trigger a reload of the driver to ensure that the settings have taken effect.

Please note that this tool will need to be run on a system before using it to run the SDK samples or test an application using NVFBC.

4.1.4 Enabling generation of textual logs

NVFBC and NVIFR support generating textual of varying verbosity. Logging can be enabled by setting the following registry entries:

NVIFR:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\NVIDIA Corporation\GRID]

"NVIFRLog"=dword:000000xy

[HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\NVIDIA Corporation\GRID]

"NVIFRLog"=dword:000000xy
```

NVFBC:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\NVIDIA Corporation\GRID]

"NVFBCLog"=dword:000000xy

[HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\NVIDIA Corporation\GRID]

"NVFBCLog"=dword:000000xy
```

Permissible values for x:

```
0- No logging
1- Log errors only
2- Log errors and names of APIs called
3- Log errors, names of APIs and parameters passed to APIs
4- Log everything
```

Permissible values for y:

```
0- No logging
1- Log to console
2- Log to console and text file on disk, in c:\GridLog directory.
```

In case of any problems encountered while using the NVFBC or NVIFR APIs, the client can collect the logs by setting these registry settings.