NVIDIA Driver Settings Programming Guide

Programming Guide

PG-5116-001-v01

Version	Date	Authors	Description of Change
1.0	2/19/2010	li/cc	Initial Release
1.1	6/1/2010	cc	Removed NDA requirement

Preface

Welcome to the NVIDIA Driver Settings Programming guide. This document explains how the NVIDIA set of drivers handle preferences and how these features are exposed through the NVIDIA NVAPI interface.

The NVIDIA Control Panel–provided with all NVIDIA drivers– allows the user to control the behavior of all the NVIDIA drivers in the system. But for those users who want a more in-depth control of the preferences in the NVIDIA driver, as well as for those users that wish to write their own Control Panel, this guide is for you. It explains the basic principles that constitute the NVIDIA settings framework and details how to program using NVAPI in order to retrieve and change any setting.

The NVIDIA team wishes you good luck on your endeavors!

TABLE OF CONTENTS

01 INTRODUCTION

The purchase of an NVIDIA card, whether GeForce or Quadro, opens a whole new horizon of possibilities to a computer. Advanced DirectX graphics, the latest OpenGL standard, the ability to program multicores with CUDA and OpenCL, and the ability to drive several of the largest displays on the market simultaneously are some of the new possibilities open by using an NVIDIA card.

For each new technology offered by NVIDIA there are many preferences exposed in order to make the system behave exactly how the user wants it. Examples of such preferences are the type of SLI rendering to use or whether high antialias rendering should always be used.

The NVIDIA Control Panel exposes all the preferences to the user, and allows customizing the computer to match the user's needs.

With the Release 256 NVIDIA driver all the internals of the preferences, or driver settings, have changed. NVIDIA now exports through NVAPI the interface to query and change the system driver settings. This is the same API that the NVIDIA Control Panel uses to access the system driver settings.

The first part of this document describes in detail the concepts that model the NVIDIA driver settings. The second part of this document goes in detail how to program against the NVAPI interface in order to access, query, or change the system driver settings.

02 NVIDIA DRIVER SETTINGS CONCEPTS

At any given time in a computer there are a number of programs running. More than one of those programs might be interacting with the NVIDIA driver at the same time. This section will detail the concepts upon which NVIDIA has built a system to control, on a per application basis and on a system basis, the settings selected by the user.

Driver Setting

A driver setting is any preference that affects the behavior of the NVIDIA driver. The SLI rendering mode is a driver setting. Choosing whether to sync to the vertical sync on the display is a driver setting.

As of this writing, not all driver settings are publicly exposed through the NVAPI interface. There are a number of settings that are internal to the NVIDIA driver. These settings might change from version to version of the driver.

On an installed system there are two types of settings. Those that were installed with the NVIDIA driver are called *predefined settings*. Those that were edited by the user after installation are called *user settings*.

Profile

The NVIDIA driver organizes settings in profiles. A profile is a collection of settings which can have one or more applications associated with them. A computer with an NVIDIA card has several profiles installed on it. The profiles customize the driver behavior to optimize each game and application's performance and stability.

A profile can have any number of settings, but it cannot have the same setting twice. Profiles are uniquely identified by their name.

The profiles that were shipped with the NVIDIA driver are called *predefined profiles*. The profiles that were created by the user after installation are called *user profiles*. A user profile can only contain user settings, while a predefined profile might contain user settings as well as predefined settings.

A profile can have any number of applications associated to it. A given application can only be associated with a single profile in the whole system.

A profile with one or more applications is called an *Application Profile*. Applications that have a profile have their settings applied automatically when the application starts.

The *Base Profile* is a profile that always exists and is not associated with a specific application. The settings in the Base Profile are applied to all processes on the system automatically.

A profile with no applications is called a *Global Profile*. The settings from a Global Profile are applied to all processes on the system, but only if that profile is selected to be the *Current Global Profile* in the system.

Profile Hierarchy

When an application starts, there might be three profiles contesting to provide settings to the NVIDIA driver: the *Application Profile*, the *Current Global Profile* and the *Base Profile*.

The settings from the Application Profile have the highest priority and override any settings on the other two profiles. Next in priority order comes the Current Global Profile, and finally the Base Profile.

Setting Application Behavior

The NVIDIA driver loads and applies its settings only at the time the process is initializing the NVIDIA dll. Thus, writing an application that loads the DirectX or OpenGL driver and then after that wishes to make changes to the NVIDIA Driver Settings will not obtain the desired behavior without restarting the application.

03 NVAPI PROGRAMMING GUIDE

This section provides guidance with detailed explanations and examples of how to interact with driver settings using NVAPI.

First the document provides an overview of what NVAPI is and how it is used. Then it provides some example programs and explains what each part does.

NVAPI Introduction

NVAPI is the API provided by NVIDIA to query, access, and interact with the NVIDIA driver and its capabilities.

▶ NVAPI is a DLL shipped with every NVIDIA driver and loaded dynamically by the applications using it.

The nvapi.lib used to link against is nothing more than a wrapper layer that will load the installed nvapi.dll.

▶ NVAPI is forwards and backwards compatible.

Trying to call an NVAPI function that is not implemented in the current NVAPI dll will result on **NVAPI_NO_IMPLEMENTATION**. It is easy to program and handle cases where your application is running on a system that does not support a specific function.

▶ NVIDIA provides the NVAPI SDK to download from the NVIDIA web site.

The NVAPI SDK includes the main `nvapi.h` header and `nvapi.lib` to link against. The nvapi.h header contains comments for each exported function. The NVAPI SDK from Release 256 onwards includes `NvApiDriverSettings.h`. This header contains all the NVIDIA driver settings publicly exported through NVAPI.

▶ NVAPI functions always return an **NvAPI_Status** enum.

If the function succeeds, the enum is NVAPI_OK. All the error enum are described in `nvapi.h`.

Driver Settings Code Examples

How to write a driver setting: disabling v-sync

This example program disables VSYNC for the whole system:

```c
#include "nvapi.h"
#include "NvApiDriverSettings.h"
#include <stdlib.h>
#include <stdio.h>

/*
    This function is used to print to the command line a text message
    describing the nvapi error and quits
*/
void PrintError(NvAPI_Status status)
{
    NvAPI_ShortString szDesc = {0};
    NvAPI_GetErrorMessage(status, szDesc);
    printf(" NVAPI error: %s\n", szDesc);
    exit(-1);
}

int main(int argc, char **argv)
{
    NvAPI_Status status;

    // (0) Initialize NVAPI. This must be done first of all
    status = NvAPI_Initialize();
    if (status != NVAPI_OK)
        PrintError(status);

    // (1) Create the session handle to access driver settings
    NvDRSSessionHandle hSession = 0;
    status = NvAPI_DRS_CreateSession(&hSession);
    if (status != NVAPI_OK)
        PrintError(status);

    // (2) load all the system settings into the session
    status = NvAPI_DRS_LoadSettings(hSession);
    if (status != NVAPI_OK)
        PrintError(status);

    // (3) Obtain the Base profile. Any setting needs to be inside
    // a profile, putting a setting on the Base Profile enforces it
    // for all the processes on the system
    NvDRSProfileHandle hProfile = 0;
    status = NvAPI_DRS_GetBaseProfile(hSession, &hProfile);
    if (status != NVAPI_OK)
        PrintError(status);
```

```
    // (4) Specify that we want the VSYNC disabled setting
    // first we fill the NVDRS_SETTING struct, then we call the function
    NVDRS_SETTING drsSetting = {0};
    drsSetting.version = NVDRS_SETTING_VER;
    drsSetting.settingId = VSYNCMODE_ID;
    drsSetting.settingType = NVDRS_DWORD_TYPE;
    drsSetting.u32CurrentValue = VSYNCMODE_FORCEOFF;
    status = NvAPI_DRS_SetSetting(hSession, hProfile, &drsSetting);
    if (status != NVAPI_OK)
        PrintError(status);

    // (5) Now we apply (or save) our changes to the system
    status = NvAPI_DRS_SaveSettings(hSession);
    if (status != NVAPI_OK)
        PrintError(status);

    // (6) We clean up. This is analogous to doing a free()
    NvAPI_DRS_DestroySession(hSession);
    hSession = 0;
    return 0;
}
```

This program sets the `VSYNCMODE` setting to have the value `VSYNCMODE_FORCEOFF` for the whole system.

The function `PrintError` takes an `NvAPI_Status`, retrieves from NVAPI a string describing the error, prints it on the screen and exits.

There are 6 main pieces of code, marked with the number in the comments. Following is the expanded explanation:

▶ (0) This code initializes the NVAPI interface. It makes sure that all NVAPI functions can be called. This function needs to be called before any other NVAPI functions. If this function fails, no NVAPI functionality is available.

▶ (1) This function creates a session handle. All interactions with the driver settings need a session handle.

▶ (2) The session must be loaded with all the settings on the system in order to interact with those settings.

▶ (3) As described in the concept session, the Base Profile is the profile with settings that apply to all processes in the system. Most of the time we want to update the settings in the Base Profile. This code retrieves a profile handle to the base profile.

▶ (4) This is the code that actually specifies the setting. It starts by filling a `NVDRS_SETTING` structure. The important fields to fill are:

  • `version`: This field is used by NVAPI to ensure that the size of the structure is correct.

  • `settingId`: This field specifies which setting we want to modify. The ID for the setting comes from the `nvApiDriverSettings.h` header.

- `settingType`: NVAPI supports settings of binary, dword and wstring types. The setting type needs to be specified and agree with what field of the structure holds the setting value.

- `u32CurrentValue:` This is the field to specify dword settings. The value assigned here also comes from the `nvApiDriverSettings.h` header.

▶ (5) After doing any modifications to the session as a whole, we must call the function to save the settings if we intend to actually modify the system settings. This is the call that saves the settings.

▶ (6) This last call frees any resources associated with the session. This also frees resources associated with profile handles.

How to enumerate all the profiles in the system

This program will enumerate all profiles in the system.

```
void EnumerateProfilesOnSystem()
{
    NvAPI_Status status;

    // (0) Initialize NVAPI. This must be done first of all
    status = NvAPI_Initialize();
    if (status != NVAPI_OK)
        PrintError(status);

    // (1) Create the session handle to access driver settings
    NvDRSSessionHandle hSession = 0;
    status = NvAPI_DRS_CreateSession(&hSession);
    if (status != NVAPI_OK)
        PrintError(status);

    // (2) load all the system settings into the session
    status = NvAPI_DRS_LoadSettings(hSession);
    if (status != NVAPI_OK)
        PrintError(status);

    // (3) Enumerate through all the profiles. Do so until the
    // return value is NVAPI_END_ENUMERATION.
    NvDRSProfileHandle hProfile = 0;
    unsigned int index = 0;
    while((status = NvAPI_DRS_EnumProfiles(hSession, index, &hProfile))
== NVAPI_OK) {
        // (4) hProfile is now a valid Profile handle.
        // retrieve information from the Profile
        printf("Profile in position %d:\n", index);
        DisplayProfileContents(hSession, hProfile);
        index++;
    }
    if (status == NVAPI_END_ENUMERATION) {
        // this is expected at the end of the enumeration
    } else if (status != NVAPI_OK)
        PrintError(status);

    // (6) We clean up. This is analogous to doing a free()
    NvAPI_DRS_DestroySession(hSession);
    hSession = 0;
}
```

Points (0) through (2) and (6) are the same as in the previous example.

Point (3) is how Profile enumeration works. It is possible to obtain the total number of profiles in the system or, as the example code does, keep calling the function incrementing the index for the profile to retrieve until we obtain the NVAPI status that indicates that there are not more profiles.

How to display all information related to a profile

This code is a function that assumes we have a valid session and profile handles. It will retrieve all information related to a profile, its name, all associated applications, and all associated settings.

```
bool DisplayProfileContents(NvDRSSessionHandle hSession,
                            NvDRSProfileHandle hProfile)
{
    // (0) this function assumes that the hSession and hProfile are
    // valid handles obtained from nvapi.
    NvAPI_Status status;

    // (1) First, we retrieve generic profile information
    // The structure will provide us with name, number of applications
    // and number of settings for this profile.
    NVDRS_PROFILE profileInformation = {0};
    profileInformation.version = NVDRS_PROFILE_VER;
    status = NvAPI_DRS_GetProfileInfo(hSession,
                                      hProfile,
                                      &profileInformation);

    if (status != NVAPI_OK) {
        PrintError(status);
        return false;
    }

    wprintf(L"Profile Name: %s\n", profileInformation.profileName);
    printf("Number of Applications associated with the Profile: %d\n",
            profileInformation.numOfApps);
    printf("Number of Settings associated with the Profile: %d\n",
            profileInformation.numOfSettings);
    printf("Is Predefined: %d\n", profileInformation.isPredefined);


    // (2) Now we enumerate through all the applications on the profile,
    // if there is any
    if (profileInformation.numOfApps > 0) {
        NVDRS_APPLICATION appArray =
            new NVDRS_APPLICATION[profileInformation.numOfApps];
        unsigned int numAppsRead = profileInformation.numOfApps, i;
        // (3) It is possible to enumerate all applications one by one,
        // or all at once on a preallocated array. The numAppsRead
        // represents the number of NVDRS_APPLICATION structures
        // allocated in the array. It will be modified on return of the
        // function  contain the number of actual applications that have
        // been filled by NVAPI
        appArray[0].version = NVDRS_APPLICATION_VER;
        status = NvAPI_DRS_EnumApplications(hSession,
                                            hProfile,
                                            0,
                                            &numAppsRead,
                                            appArray);
        if (status != NVAPI_OK) {
```

```
        PrintError(status);
        delete[] appArray;
        return false;
    }
    for(i=0; i<numAppsRead; i++) {
        wprintf(L"Executable: %s\n", appArray[i].appName);
        wprintf(L"User Friendly Name: %s\n",
            appArray[i].userFriendlyName);
        printf("Is Predefined: %d\n", appArray[i].isPredefined);
    }
    delete[] appArray;
}

// (4) Now we enumerate all the settings on the profile
if (profileInformation.numOfSettings > 0) {
    NVDRS_SETTING setArray =
        new NVDRS_SETTING[profileInformation.numOfSettings];
    unsigned int numSetRead = profileInformation.numOfSettings, i;
    // (5) The function to retrieve the settings in a profile works
    // like the function to retrieve the applications.
    setArray[0].version = NVDRS_SETTING_VER;
    status = NvAPI_DRS_EnumSettings(hSession,
                                    hProfile,
                                    0,
                                    &numSetRead,
                                    setArray);

    if (status != NVAPI_OK) {
        PrintError(status);
        return false;
    }
    for(i=0; i<numSetRead; i++) {
        if (setArray[i].settingLocation !=
                NVDRS_CURRENT_PROFILE_LOCATION) {
        // (6) The settings that are not from the Current Profile
        // are inherited from the Base or Global profiles. Skip them.
            continue;
        }
        wprintf(L"Setting Name: %s\n", setArray[i].settingName);
        printf("Setting ID: %X\n", setArray[i].settingId);
        printf("Predefined? : %d\n",
        setArray[i].isCurrentPredefined);
        switch (setArray[i].settingType) {
            // (7) a setting can be of different types and be using
            // different fields on the NVDRS_SETTING union
            case NVDRS_DWORD_TYPE:
                printf("Setting Value: %X\n",
                    setArray[i].u32CurrentValue);
                break;
            case NVDRS_BINARY_TYPE:
                {
                    unsigned int len;
                    printf("Setting Binary (length=%d) :",
                        setArray[i].binaryCurrentValue.valueLength);
                    for(len=0;
```

```
                        len<setArray[i].binaryCurrentValue.valueLength;

                            len++)
                        {
                            printf(" %02x",
                         setArray[i].binaryCurrentValue.valueData[len]);
                        }
                        printf("\n");
                    }
                    break;
                case NVDRS_WSTRING_TYPE:
                    wprintf(L"Setting Value: %s",
                        setArray[i].wszCurrentValue);
                    break;
            }
        }
    }
    printf("\n");
    return true;
}
```

▶ (1) This code retrieves the generic Profile information. This contains a name, the number of applications and settings, and whether this profile is predefined or not.

▶ (2) This if statement contains the code to enumerate all applications. For each application, we retrieve the name of the executable, the Friendly name and whether it was added as a predefined application or not.

▶ (3) Enumerating applications can be done the same way profiles were enumerated in the previous example. In this case, we use the second way of enumerating all applications. We know how many applications are there, we allocate an array to hold information for all applications and with a single call we obtain them all.

▶ (4) In the same way we enumerated applications, this code enumerates settings.

▶ (5) The function to enumerate settings follows the same footprint as the function to enumerate application.

▶ (6) When settings are enumerated you obtain all settings for a profile, even the inherited settings. Skip the settings that do not belong to the requested profile.

04 NVIDIA DRIVER SETTINGS - BEST PROGRAMMING PRACTICES

The section contains a few suggestions and general warnings to consider when programming with NVAPI for the NVIDIA Driver Settings.

▶ NVIDIA advises loading Settings at the beginning of the program and save them only at the end of the program. The session can hold all of your current changes.

▶ Processes only load their settings at loading time. Save the settings to disk before a new process runs in order to have it use the new settings.

▶ You can have as many Sessions open as you want. If another process modifies the NVIDIA driver settings while your process has an open session, you will not be notified. Saving the session and loading it again is a good way to update your session.

▶ There are functions to reset the state of a setting for all settings in a Profile as well as for the whole system. Each individual function restores the state of a setting, profile, or the system to what amounts to a clean driver install. The functions are:

- NvAPI_DRS_RestoreProfileDefaultSetting()
- NvAPI_DRS_RestoreProfileDefault()
- NvAPI_DRS_RestoreAllDefaults()

> **!** **CAUTION:** On Release R256 Session merging is not implemented. This means that when you save your session settings, if another process had modified the system settings, their modifications will be lost. This issue affects systems where multiple processes are trying to modify settings at the same time. Please advise your users to run your application modifying settings alone, without the NVIDIA Control Panel opened in the background. This limitation is expected to be removed in a future release.

05 NVAPI FUNCTION DOCUMENTATION

The full documentation of all NVAPI functions is the `nvapi.h` header itself. The comments in the header are the most up to date documentation for the NVAPI functions. Here we list the NVAPI functions used in the first code example.

NvAPI_Initialize()

The function `NvAPI_Initialize` needs to be called before any other NVAPI function is called. This function can be called several times safely.

```
NVAPI_INTERFACE NvAPI_Initialize();
```

NvAPI_GetErrorMessage()

The function `NvAPI_GetErrorMessage` provides a string describing the error from an NvAPI_Status enum. The string is returned in the szDesc parameter.

```
NVAPI_INTERFACE NvAPI_GetErrorMessage(NvAPI_Status nr,
                                      NvAPI_ShortString szDesc);
```

NvAPI_DRS_CreateSession()

All interactions with the NVIDIA driver settings requires a `NvDRSSessionHandle`. This is a handle that represents the current interaction with NVAPI and the driver settings. The function `NvAPI_DRS_CreateSession` creates this handle and must be called to retrieve a handle to use on all subsequent NVAPI calls.

```
NVAPI_INTERFACE NvAPI_DRS_CreateSession(NvDRSSessionHandle *phSession);
```

NvAPI_DRS_DestroySession()

The function **NvAPI_DRS_DestroySession** destroys the session handle. After this function call, the session handle is not valid anymore and should not be used. Failure to call this function might result in memory leaks.

```
NVAPI_INTERFACE NvAPI_DRS_DestroySession(NvDRSSessionHandle hSession);
```

NvAPI_DRS_LoadSettings()

The function **NvAPI_DRS_LoadSettings** loads all the system settings on the specified session. The settings from the system are not loaded until this function is called. In order to browse settings or to set settings this function needs to be called.

```
NVAPI_INTERFACE NvAPI_DRS_LoadSettings(NvDRSSessionHandle hSession);
```

NvAPI_DRS_SaveSettings()

The function **NvAPI_DRS_SaveSettings** saves all modifications done to the session passed as a parameter to the system. Until this function is called, nothing has been modified on the system. If this function is not called, then no modifications are performed on the system.

```
NVAPI_INTERFACE NvAPI_DRS_SaveSettings(NvDRSSessionHandle hSession);
```

NvAPI_DRS_GetBaseProfile()

The function **NvAPI_DRS_GetBaseProfile** retrieves a handle to the Base Profile in the system. This handle can be used to interact with the profile by adding, removing, or iterating through the settings in the profile.

```
NVAPI_INTERFACE NvAPI_DRS_GetBaseProfile(NvDRSSessionHandle hSession,
                                         NvDRSProfileHandle *phProfile);
```

NvAPI_DRS_SetSetting()

The function **NvAPI_DRS_SetSetting** specifies the value for a given setting on a given profile. If the setting exists, the previous value is overwritten.

This function does not delete a setting. To delete a setting, call **NvAPI_DRS_DeleteProfileSetting**.

```
NVAPI_INTERFACE NvAPI_DRS_SetSetting(NvDRSSessionHandle hSession,
                                     NvDRSProfileHandle hProfile,
                                     NVDRS_SETTING *pSetting);
```

www.nvidia.com