NeroAPI

v7.0.5.6

The *NeroAPI* will only work with
a fully installed Nero version!

Document version 1.70, Copyright 2002-2006 Nero AG

Nero AG, Im Stoeckmaedle 18, 76307 Karlsbad, Germany

1. Contents

1. Contents ... 2
2. License Agreement ... 9
3. Introduction ... 10
3.1. Motivation ... 10
3.2. Overview ... 10
3.3. Requirements ... 10
3.4. Required Skills ... 11
3.5. Compatibility between Different NeroAPI Versions ... 11
3.5.1. Source Compatibility ... 11
3.5.2. Binary Compatibility ... 11
3.6. Related Topics ... 11
3.7. The NeroSDK Forum ... 11
4. Quick Start ... 12
4.1. Running Precompiled Sample Applications ... 12
4.1.1. NeroAPITest ... 12
4.1.2. NeroAPITest Command Line Examples ... 14
4.2. Compiling the Samples ... 15
4.3. Accessing the *NeroAPI* in Your Applications ... 15
4.4. Points of Consideration ... 15
5. Detailed Discussion of the NeroAPITest Sample ... 16
6. Creating a Simple MFC Application ... 19
6.1. Nero Fiddled While Rome Burned! ... 19
6.2. Creating the Framework ... 19
6.3. Adding NeroAPI files ... 21
6.4. Adding Controls ... 22
6.5. Adding Member Variables ... 24
6.5.1. Variables for Controls ... 24
6.5.2. Other Variables ... 25
6.6. Adding Message Handling Functions For Controls ... 26
6.6.1. OnBrowse ... 26
6.6.2. OnRemove ... 27
6.6.3. OnSelchangeFilelist ... 28
6.6.4. OnBurn ... 28
6.6.5. OnOK ... 34
6.6.6. OnCancel ... 34
6.6.7. OnAbort ... 34
6.6.8. OnAudio ... 35
6.6.9. OnImport ... 35
6.7. Adding Utility Functions ... 35
6.7.1. NeroAPIInit ... 35
6.7.2. NeroAPIFree ... 38
6.7.3. AppendString ... 39
6.7.4. GetFilename ... 39
6.7.5. MergeIsoTracks ... 39
6.7.6. DeleteIsoItemTree ... 41
6.7.7. FreeOurOwnResources ... 41

6.8. Adding Callback Functions 42
6.8.1. IdleCallback 42
6.8.2. UserDialog 43
6.8.3. ProgressCallback 45
6.8.4. AbortedCallback 45
6.8.5. AddLogLine 45
6.8.6. SetPhaseCallback 46
6.9. Build and Run NeroFiddles 46
7. API Types and Functions 47
7.1. Types 47
7.1.1. DLG_OVERBURN_INFO 47
7.1.2. NERO_ABORTED_CALLBACK 47
7.1.3. NERO_ACCESSTYPE 48
7.1.4. NERO_ADD_LOG_LINE_CALLBACK 48
7.1.5. NERO_AUDIO_FORMAT_INFO 48
7.1.6. NERO_AUDIO_ITEM_INFO 49
7.1.7. NERO_AUDIO_ITEM_HANDLE 49
7.1.8. NERO_AUDIO_TRACK 49
7.1.9. NERO_CALLBACK 50
7.1.10. NERO_CD_COPY 51
7.1.11. NERO_CD_FORMAT 53
7.1.12. NERO_CD_INFO 54
7.1.13. NERO_CITE_ARGS 56
7.1.14. NERO_CONFIG_RESULT 57
7.1.15. NERO_DATA_EXCHANGE 58
7.1.16. NERO_DATA_EXCHANGE_TYPE 59
7.1.17. NERO_DEVICEHANDLE 59
7.1.18. NERO_DISABLE_ABORT_CALLBACK 60
7.1.19. NERO_DEVICEOPTION 60
7.1.20. NERO_DLG_WAITCD_MEDIA_INFO 61
7.1.21. NERO_DRIVE_ERROR 62
7.1.22. NERO_DRIVESTATUS_CALLBACK 63
7.1.23. NERO_DRIVESTATUS_TYPE 63
7.1.24. NERO_DRIVESTATUS_RESULT 64
7.1.25. NERO_FILESYSTEMTRACK_OPTIONS 64
7.1.26. NERO_FREESTYLE_TRACK 65
7.1.27. NERO_FS_NAMETYPE 66
7.1.28. NERO_FS_TYPE 66
7.1.29. NERO_IDLE_CALLBACK 67
7.1.30. NERO_IMPORT_DATA_TRACK_INFO 67
7.1.31. NERO_IMPORT_DATA_TRACK_RESULT 68
7.1.32. NERO_IO 68
7.1.33. NERO_IO_CALLBACK 69
7.1.34. NERO_ISO_ITEM 69
7.1.35. NERO_MAJOR_PHASE 70
7.1.36. NERO_SET_MAJOR_PHASE_CALLBACK 72
7.1.37. NERO_MEDIA_SET 72
7.1.38. NERO_MEDIA_TYPE 73

7.1.39. NERO_MEDIUM_TYPE ... 75
7.1.40. NERO_BOOK_TYPE ... 76
7.1.41. NERO_PROGRESS ... 77
7.1.42. NERO_PROGRESS_CALLBACK ... 77
7.1.43. NERO_SCSI_DEVICE_INFO ... 78
7.1.44. NERO_SCSI_DEVICE_INFOS ... 81
7.1.45. NERO_SET_PHASE_CALLBACK ... 82
7.1.46. NERO_SETTINGS ... 82
7.1.47. NERO_SPARE_AREA_SETTINGS ... 83
7.1.48. NERO_SPEED_INFOS ... 84
7.1.49. NERO_STATUS_CALLBACK ... 84
7.1.50. NERO_TEXT_TYPE ... 84
7.1.51. NERO_TRACK_INFO ... 86
7.1.52. NERO_TRACK_TYPE ... 87
7.1.53. NERO_TRACK_FS_TYPE ... 87
7.1.54. NERO_TRACKMODE_TYPE ... 88
7.1.55. NERO_VIDEO_ITEM_TYPE ... 88
7.1.56. NERO_VIDEO_ITEM ... 89
7.1.57. NERO_VIDEO_RESOLUTION ... 89
7.1.58. NERO_VMS_INFO ... 90
7.1.59. NERO_VMSSESSION ... 90
7.1.60. NERO_WAITCD_TYPE ... 91
7.1.61. NERO_WRITE_CD ... 94
7.1.62. NERO_WRITE_FILE_SYSTEM_CONTENT ... 95
7.1.63. NERO_WRITE_FREESTYLE_CD ... 96
7.1.64. NERO_WRITE_IMAGE ... 98
7.1.65. NERO_WRITE_VIDEO_CD ... 99
7.1.66. NERO_UDF_REVISION ... 100
7.1.67. NERO_UDF_PARTITION_TYPE ... 100
7.1.68. NEROAPI_BURN_ERROR ... 101
7.1.69. NEROAPI_OPTION ... 101
7.1.70. NEROAPI_INIT_ERROR ... 103
7.1.71. NEROAPI_SCSI_DEVTYPE ... 103
7.1.72. NERODLG_ICON_TYPE ... 104
7.1.73. NERODLG_MESSAGE_TYPE ... 104
7.1.74. NERODLG_MESSAGEBOX ... 105
7.1.75. NeroUserDlgInOutEnum ... 105
7.1.76. ROBOMOVEMESSAGE ... 110
7.1.77. ROBOMOVENODE ... 110
7.1.78. ROBOUSERMESSAGE ... 111
7.1.79. ROBOUSERMESSAGETYPE ... 111
7.2. Functions ... 112
7.2.1. NeroAudioCreateTargetItem ... 112
7.2.2. NeroAudioCloseItem ... 112
7.2.3. NeroAudioGetFormatInfo ... 112
7.2.4. NeroAudioGUIConfigureItem ... 112
7.2.5. NeroBurn ... 113
7.2.6. NeroClearErrors ... 114

7.2.7. NeroCloseDevice ... 114
7.2.8. NeroConvertToValidFileSystemName ... 115
7.2.9. NeroCopyIsoItem ... 115
7.2.10. NeroCreateIsoItem ... 116
7.2.11. NeroCreateIsoItemOfSize ... 116
7.2.12. NeroCreateIsoTrackEx ... 116
7.2.13. NeroCreateProgress ... 118
7.2.14. NeroDAE ... 118
7.2.15. NeroDone ... 119
7.2.16. NeroEjectLoadCD ... 119
7.2.17. NeroEraseCDRW ... 119
7.2.18. NeroEraseDisc ... 120
7.2.19. NeroEstimateTrackSize ... 120
7.2.20. NeroFreeCDStamp ... 121
7.2.21. NeroFreeIsoItem ... 121
7.2.22. NeroFreeIsoTrack ... 122
7.2.23. NeroFreeIsoItemTree ... 122
7.2.24. NeroFreeMem ... 122
7.2.25. NeroGetAPIVersion ... 123
7.2.26. NeroGetAPIVersionEx ... 123
7.2.27. NeroGetAvailableDrivesEx ... 123
7.2.28. NeroGetAvailableSpeeds ... 124
7.2.29. NeroGetCDInfo ... 124
7.2.30. NeroGetCDRWErasingTime ... 125
7.2.31. NeroGetCurrentMediumType ... 126
7.2.32. NeroGetDeviceOption ... 126
7.2.33. NeroGetDiscErasingTime ... 127
7.2.34. NeroGetDiscImageInfo ... 127
7.2.35. NeroGetErrorLog ... 128
7.2.36. NeroGetLastDriveError ... 128
7.2.37. NeroGetLastError ... 129
7.2.38. NeroGetLastErrors ... 129
7.2.39. NeroGetLocalizedWaitCDTexts ... 129
7.2.40. NeroGetTypeNameOfMedia ... 130
7.2.41. NeroGetVMSInfo ... 130
7.2.42. NeroGetWaitCDTexts ... 131
7.2.43. NeroImportDataTrack ... 131
7.2.44. NeroImportIsoTrackEx ... 132
7.2.45. NeroInit ... 133
7.2.46. NeroInitImageRecorder ... 133
7.2.47. NeroIsDeviceReady ... 133
7.2.48. NeroOpenDevice ... 134
7.2.49. NeroRegisterDriveChangeCallback ... 134
7.2.50. NeroRegisterDriveStatusCallback ... 135
7.2.51. NeroSetDesiredUDFRevision ... 135
7.2.52. NeroSetDesiredUDFPartitionType ... 136
7.2.53. NeroSetDeviceOption ... 136
7.2.54. NeroSetExpectedAPIVersion ... 137

7.2.55. NeroSetExpectedAPIVersionEx ... 137
7.2.56. NeroSetOption ... 138
7.2.57. NeroUpdateDeviceInfo ... 138
7.2.58. NeroUnregisterDriveChangeCallback ... 138
7.2.59. NeroUnregisterDriveStatusCallback ... 139
7.2.60. NeroUserDlgInOut ... 139
7.2.61. NeroWaitForDeviceReady ... 140
7.2.62. NeroWaitForDisc ... 140
7.2.63. NeroWaitForMedia ... 141
8. ISO Track Creation ... 142
9. ISO Track Classes ... 143
9.1. Overview ... 143
9.2. CNeroDataCallback ... 144
9.3. CNeroIsoHandle ... 144
9.4. CNeroIsoIterator ... 145
9.5. CNeroIsoEntry ... 145
9.6. CNeroIsoTrack ... 147
10. The FileSystemContent Interface ... 150
10.1. Overview ... 151
10.2. Namespace setting ... 152
10.3. InterfaceBase ... 152
10.4. File System Reading Interfaces ... 153
10.4.1. IFileContent ... 153
10.4.2. IDirectoryEntry ... 153
10.4.3. IDirectory ... 154
10.4.4. IFileSystemContent ... 154
10.5. File System Content Creation Interfaces ... 155
10.5.1. IDataInputStream ... 155
10.5.2. IFileProducer ... 155
10.5.3. IDirectoryEntryContainer ... 155
10.5.4. IDirectoryContainer ... 156
10.5.5. IFileSystemDescContainer ... 157
10.5.6. IFileSystemDescContainerVolume ... 157
11. The Burn-at-once Interface ... 159
11.1. The NERO_WRITE_BURN_AT_ONCE struct ... 159
11.2. The IBurnAtOnceInfo Interface ... 160
11.2.1. GetOffset ... 160
11.3. Functions ... 161
11.3.1. NeroBurnAtOnce ... 161
11.3.2. NeroBAOCreateHandle ... 161
11.3.3. NeroBAOOpenFile ... 163
11.3.4. NeroBAOWriteToFile ... 163
11.3.5. NeroBAOCloseFile ... 164
11.3.6. NeroBAOCloseHandle ... 164
12. The Packet Writing API ... 165
12.1. Packet Writing Interface ... 165
12.1.1. Access Mode ... 165
12.1.2. ImageAccessMode ... 165

12.1.3. NeroCreateBlockWriterInterface ... 166
12.1.4. NeroCreateBlockReaderInterface ... 166
12.1.5. NeroCreateBlockAccessFromImage ... 166
12.1.6. NeroGetSupportedAccessModesForDevice ... 166
12.2. File System Block Access Interface ... 167
12.2.1. INeroFileSystemBlockAccess ... 168
12.2.2. INeroFileSystemBlockAccessExtension ... 168
12.2.3. INeroFileSystemBlockReader ... 169
12.2.4. INeroFileSystemBlockWriter ... 170
12.2.5. InterfaceType ... 171
12.2.6. NeroFSBlockAccessExtensionsType ... 171
12.2.7. NeroFSError ... 171
12.2.8. NeroFSPartitionInfo ... 172
12.2.9. NeroFSTrackType ... 173
12.2.10. NeroFSSecNo ... 173
13. Robot Control Interface ... 174
13.1. NERO_COMMNODE_TYPE Enumeration ... 174
13.2. NEROAPI_ROBO_ERROR Enumeration ... 174
13.3. NeroPrintLabelCallback_t Callback ... 174
13.4. NERO_ROBO_DRIVER_INFO Structure ... 175
13.5. NERO_ROBO_DRIVER_INFOS Structure ... 175
13.6. NeroGetAvailableRoboDrivers Function ... 175
13.7. NeroAssociateRobo Function ... 175
13.8. NERO_ROBO_FLAG Enumeration ... 176
13.9. NeroSetRoboFlag Function ... 176
14. Media Type Formats ... 177
14.1. Audio ... 177
14.2. Video ... 177
14.2.1. SVCD Creation with *Nero* ... 177
15. FAQ ... 179
15.1. NeroSDK License ... 179
15.2. NeroSDK/NeroAPI Features ... 179
15.3. General Programming Issues ... 180
15.4. DVD Issues ... 180
15.5. Video CD and Super Video CD ... 182
15.6. Audio ... 183
15.7. Multisession ... 183
15.8. Size Information, Calculation and Estimation ... 185
15.9. Packet Writing ... 186
15.10. Concurrency ... 186
15.11. Miscellaneous ... 187
16. Known Limitations ... 191
17. Bibliography ... 192
17.1. C Programming Books ... 192
17.2. C Programming Online Resources ... 192
17.3. C++ Programming Books ... 192
17.4. C++ Online Resources ... 193
17.5. General CD/CD-ROM Online Resources ... 193

17.6. Audio CD Online Resources 193
17.7. Super Video CD Online Resources 193

2. License Agreement

IMPORTANT: PLEASE READ THE SOFTWARE LICENSE AGREEMENT ("LICENSE") CAREFULLY BEFORE USING THE SOFTWARE.

USING THE SOFTWARE INDICATES YOUR ACKNOWLEDGMENT THAT YOU HAVE READ THE LICENSE AND AGREE TO ITS TERMS.

The license agreement is contained in a text file, “NeroSDK_License.txt”, to be found in the root folder of the installation package.

3. Introduction

3.1. Motivation

The *NeroSDK* is a tool for inclusion of *Nero* functionality in your own applications.

Since it became available we have provided documentation in form of source code comments and a read-me-file. That kind of documentation proved as being ample for seasoned professionals, though somewhat tedious. Soon more and more people began to use this SDK. It became obvious that a “manual within the code” was not enough for programmers who are less familiar with getting the grip on somebody else’s source code.

You asked for this documentation, we have created it. We hope this little manual, which describes the *NeroAPI* part of the *NeroSDK*, to be just what you need.

Unfortunately nothing is ever perfect. So the author would be grateful if you sent your suggestions or pointed out errors, both in our code and documentation.

3.2. Overview

This paper, the documentation of the *NeroAPI*, contains some practical guidelines on how to use the API (Application Programming Interface) that is implemented in the NeroAPI.dll with the help of *NeroSDK* (Nero Software Development Kit). The SDK is available for OEMs (Original Equipment Manufacturers) and registered users of *Nero*.

We believe that it will help you add CD- and DVD-burning capability to your own applications in less time. A detailed discussion of what the programming samples do, and how they do it, together with a list of all types and functions, should enable you to get your routines working in no time.

A brief description of *Nero’s* Audio-, Video- and Super-Video-CD capabilities will ensure that you do not fail because of using the wrong file format.

3.3. Requirements

This documentation assumes that *Nero* 7.0.8.2 or later is already installed on your computer. The current *NeroSDK* version (*NeroSDK* 1.08) supports the command set of *Nero* 7.0.8.2. The version numbers of *Nero* and *NeroAPI* are not synchronized. Therefore, you should not assume that a certain *Nero* version will fully support a *NeroAPI* release that has an identical version label, unless expressly stated.

Depending on your individual needs, your *NeroAPI*-based application might work with an earlier version. Therefore, this documentation will indicate which *NeroAPI* version introduced a particular feature, function, or type.

For additional information please take a look at *3.5 Compatibility between Different NeroAPI Versions*.

3.4. Required Skills

This documentation is directed towards Software developers who have gathered some experience in programming C or C++. It is absolutely required that you know the basic concepts of the C programming language to use the *NeroAPI*.

If you have no or little experience with C/C++, you will find a list of books and Internet addresses that we regard as very useful for learning the language. C and C++ are still the most commonly used programming languages, and once you've mastered them you will learn any other programming language with ease.

3.5. Compatibility between Different NeroAPI Versions

Since several programs must access the same *NeroAPI* package, not all of them can be updated every time a new version of *Nero* is released.

Nero ensures both source and binary compatibility with its future versions. Version 5.0.3.9 of *Nero* and the following versions meet that requirement, while older versions do not.

3.5.1. Source Compatibility

Applications written for one version of the *NeroAPI* will work with more recent versions of *Nero*, without having to change their source code.

3.5.2. Binary Compatibility

Applications written for one version of the *NeroAPI* will work with more recent versions of *Nero* without having to compile the application again.

3.6. Related Topics

Closely related to *NeroAPI* is *NeroCOM*, a Type Library for the Component Object Model. *NeroCOM* presents another approach for accessing the power of *Nero*.

NeroCOM will be installed as part of the *Nero* installation. The documentation for *NeroCOM* is available as part of the *NeroSDK*.

3.7. The NeroSDK Forum

We provide a forum for all users of the *NeroSDK* to get in dialog with each other at http://www.nero.com/link.php?topic_id=17. We will also monitor the messages from time to time and try to help where possible.

4. Quick Start

4.1. Running Precompiled Sample Applications

4.1.1. NeroAPITest

This application can

- read information about a CD
- burn audio CDs
- burn ISO CDs
- burn UDF (Universal Disc Format) CDs
- burn ISO/UDF CDs
- burn DVDs
- burn mixed mode CDs
- burn Video and Super Video CD
- extract CDA tracks

Open a command window (MS-DOS shell) and type “NeroAPITest“, followed by a command and in most cases a parameter list.

See the following table for valid parameters. Square brackets indicate that a parameter is optional. However, when writing an Audio/ISO CD, you have to supply at least one valid set of parameters.

Command	**Function**	
--listdrives	List available drives	
	Parameters	**Description**
	None	
Command	**Function**	
--cdinfo	Get information about a CD	
	Parameters	**Description**
	--drivename ‘x’	Supply drive letter.
Command	**Function**	
--write	Write Audio/ISO/UDF/Mixed Mode CD or DVD	
	Parameters	**Description**
	--drivename 'x'	Supply drive letter.
	[--real]	Do not simulate burning process.
	[--TAO]	Track At Once.
	[--bup]	Burn with buffer underrun protection.
	[--writebuffersize ‘x’]	Set the size of the write buffer in Kbytes.

Command	Function	
	[--artist 'artist']	Supply artist's name for Audio CD.
	[--title 'title']	Supply Audio CD title.
	[--speed 'x']	Select desired speed.
	[--pause x]	Pause length in blocks. Valid pause lengths are 0 to 7500. In TAO only default pause length is supported (150).
	[--audioinindex0]	Write audio data into index 0 to prevent silent pauses between tracks. TAO and –audioinindex0 are mutually exclusive.
	['audio file1'] ['audio file2'] ...	List of Audio files to burn.
	[--cdextra]	Use the CDExtra feature. Two sessions will be created, the first containing Audio tracks, the second containing one ISO track.
	[--iso 'volume name']	Name the ISO-volume.
	[--udf 'volume name']	UDF
	[--isoudf 'volume name']	ISO and UDF
	[--dvd]	Burn ISO DVD
	[--iso-no-joliet]	No long filenames.
	[--iso-mode2]	Select ISO mode 2.
	['dir/file1'] ['dir/file2'] ...	List of files to burn. Can be directory tree or file.
--write	Write Video CD	
	Parameters	**Description**
	--drivename 'xxx'	Supply drive letter.
	--videocd	Selection of Video CD type
	[--real]	Do not simulate burning process.
	[--TAO]	Track At Once.
	[--bup]	Burn with buffer underrun protection.
	[--writebuffersize 'x']	Set the size of the write buffer in Kbytes.
	[--speed 'x']	Select desired speed.
	['mpeg/jpeg file1'] *['mpeg/jpeg file2']…*	List of Video files.
Command	**Function**	
--write	Write Super Video CD	
	Parameters	**Description**
	--drivename 'xxx'	Supply drive letter.
	--svideocd	Selection of Video CD type
	[--real]	Do not simulate burning process.
	[--TAO]	Track At Once.
	[--bup]	Burn with buffer underrun protection.

Command	Function	
	[--writebuffersize 'x']	Set the size of the write buffer in Kbytes.
	[--speed 'x']	Select desired speed.
	'mpeg/jpeg file1' ['mpeg/jpeg file2']…	List of Video files to burn.
Command	**Function**	
--read	Copy CD tracks to files	
	Parameters	**Description**
	-- drivename 'x'	Supply drive letter.
	-- 'xy' 'file1' [-- 'xy' 'file2']…	Read contents of track with number 'xy' into 'file 1'. The file name has to include the suffix. Only ".pcm" and ".wav" will be accepted.
Command	**Function**	
--erase	Erase a CD-RW	
	Parameters	**Description**
	[--entire]	By default a quick erase is done, where the actual content of the CD is not erased. The "—entire" option sweeps the whole CD, insuring that no data can be retrieved by any means afterwards.
	-- drivename 'x'	Supply drive letter.
--eject	Ejects a CD from the drive	
	Parameters	**Description**
	-- drivename 'x'	Supply drive letter.
Command	**Function**	
--load	Loads a CD into the drive	
	Parameters	**Description**
	-- drivename 'x'	Supply drive letter.

4.1.2. NeroAPITest Command Line Examples

Simulate the burning of a mixed mode CD with one audio track and one file using the CD recorder with the drive letter "D":

NeroAPITest --write --drivename d c:\media\audio\police01.wav --iso mycd c:\data\file.dat

Burn the content of a folder:

NeroAPITest --write --drivename d --real --iso mycd c:\data

4.2. Compiling the Samples

- Start Visual C++.
- Select *Files* and *Open Workspace* from the menu.
  An *Open*-dialog will come up.
- Select *project files* as file type (".dsp"-suffix)
- Navigate to the samples directory and select the project you want to open.
- Click the *OK*-button.
- Open the *Build*-menu and then select *Build All*.

4.3. Accessing the *NeroAPI* in Your Applications

- Make the *.h files of the *NeroAPI*-include-directory accessible from your program
- Link your project with the NeroAPIGlue library
- Use the *NeroAPI* functions in proper order. Take the samples for reference.
- Implement the callback functions.

4.4. Points of Consideration

Make sure that the application will find the required DLLs, by installing *Nero*.

The NeroInit and NeroDone functions must not be called from the destructor of a global object or from a DllMain function. If they are called like that, the result will be a deadlock.

To burn WMA files onto CD, *Nero's* WMA Plug-in must be installed.

Users of *Nero* can download the WMA Plug-in free of charge from http://www.nero.com.

Important note on the naming of identifiers

In the beginning of the *NeroAPI*, DVD writers were not widely available. The naming of the older *NeroAPI* types and constants still reflects that situation when CDs were the only available media. For example, NBF_DETECT_NON_EMPTY_CDRW seems to refer to CD-RW media only. However, it will also be accepted by the *NeroAPI* in the context of rewritable DVD media.

To remain compatible with existing implementations, we have not updated the old type names. **Unless explicitly stated otherwise, it is perfectly legal to use types or constants that contain "CD" in their name with other media, like DVDs.**

When we started supporting DVDs, we began to use the more generic term "disc".

5. Detailed Discussion of the NeroAPITest Sample

NeroAPITest,cpp is the source file that defines what the application actually does. It contains the main function which is the application's entry point. Having a main function identifies it as console application (as opposed to a true Windows application that has a winmain function).

At the beginning of NeroAPITest.cpp, you will find a few include-directives. Those are references to other files that contain some required definitions: NeroAPIGlue.h is responsible for attaching NeroAPITest to the NeroAPI.DLL.

The rest of the include files are required for communicating with your Operating System. They let you access device-Input/Output, handle special keystroke combinations like Ctrl-C (which will cause the current task to abort), display characters on your computer screen, and perform many other important background tasks.

Line 43: From line 43 on you will find function declarations. All listed functions are implemented within NeroAPITest.cpp. Most of them are callback-functions, and thus responsible for giving *NeroAPI* access to NeroAPITest whenever necessary, e.g. to display a progress bar or obtain user input.

Line 64: Beginning with line 64, types and variables are defined, and termination behavior is implemented by a sequence of *NeroAPI* calls in the Exit function, starting in line 115. The order of those API calls is very important, and if some API functions that free memory were not called, the application would allocate memory and not free it when terminating.

Line 133: The function Usage will display a help-text, concerning the proper usage of program arguments or command line parameters.

Line 158: The function main starts by initializing variables. It then parses command line parameters to determine what the user wants to do. According to those parameters variables are set, indicating what kind of device is to be used, whether the application should read or write data, what kind of data, and where the data comes from.

Line 440: The function signal tells Windows to call our SigCtrlC function when Ctrl-C is pressed.

Then the application tries to load the NeroAPI.dll by calling NeroAPIGlueConnect. If it cannot be found, an error message is displayed, and the application is terminated by calling our Exit function, providing error code 10 as a parameter. During NeroAPIGlueConnect the Windows registry is queried for the shared *NeroAPI*.

Line 448: A call to NeroGetAPIVersionEx retrieves the API's version number. The version number, contained in four WORD values, is then printed on the screen.

Line 456: The *NeroAPI* is initialized. The *NeroAPI* will work in Demo mode if no Serial Number is found.

Line 478: If the write buffer size is not 0, NeroSetOption is called to set the write buffer size in the *NeroAPI*. The actual value is calculated by taking the user-provided

parameter and multiplying it by 1024 (1 KB), because the user is expected to give the buffer size in Kbytes.

Line 483: The program then tries to get a list of all CD-ROM devices in the system by calling NeroGetAvailableDrivesEx. If no device is present, the application will terminate.

Line 490: If a drive name has been specified by the *–drivename* command, the program searches for that device in the list of available devices. The program then tries to open it for further use by calling NeroOpenDevice. If no device handle is returned (e.g. if the drive letter given is not present in the system), the application will terminate with an error message.

If *–listdrives* was passed as a command line parameter, the list of available devices will be printed.

Line 544: If a device handle could not be retrieved, the application terminates.

Line 549: This code section retrieves and displays a list of allowed speeds for the selected drive.

Line 557: Loading a CD is handled after this line.

Line 561: Erase-behavior is implemented here. The application checks whether a CD is present in the drive, whether it is of the right type, and whether the CD should undergo a quick erase or a complete sweep, to make it ready for rewriting.

Line 579: The “Eject CD” part has been implemented here.

Line 583: Lines 583 to 692 are dedicated to getting Information about a CD or reading tracks. The “read”-part after line 637 scans for all available tracks and tries to determine whether they are PCM or WAV. The respective format will be read from the CD.
If any error occurs, the application is stopped after displaying an error message.

Line 694: If the user requested to write a CD-ROM or DVD, the code after line 661 checks whether creation of ISO/Audio CD, ISO DVD, or VCD/SVCD was specified. The size of the CD/DVD directory is calculated, and the program tries to allocate memory for the NERO_WRITE_CD structure, that will be used for writing the information. If the free-memory-pool is not large enough, the application will terminate.

Then the application fills the NERO_WRITE_CD structure with the information the user provided.

Line 770: The burn process is started by calling NeroBurn, passing a pointer to the NERO_WRITE_CD structure, which has just been filled with content.

Line 784: Burning of VCD/SVCD is handled here. Basically this means filling a NERO_WRITE_VIDEO_CD structure and passing it to the NeroBurn function.

Line 840: This part of the code deals with burning an image. Allowed formats are NRG, ISO and CUE.

After the CD has been burnt, the log file is updated, the allocated memory is freed, and the application terminates without error.

Line 910: The implementation of the idle callback is empty, apart from an assertion that ensures that the *NeroAPI* actually returned the same pointer provided as user data. In a GUI application this function would probably do a little more than just to return control to the *NeroAPI* and passing the aborted-flag.

Line 931: Here the user can reply to a *NeroAPI* request by keyboard input. CharIO will take an array of mappings from character to NeroUserDlgInOut constants, and return the proper NeroUserDlgInOut constant depending on the user input.

Line 953: The UserDialog callback implementation displays a number of options, depending on the value of “type”. It then calls CharIO to get the user’s desired option. The corresponding NeroUserDlgInOut constant is then returned to the *NeroAPI*.

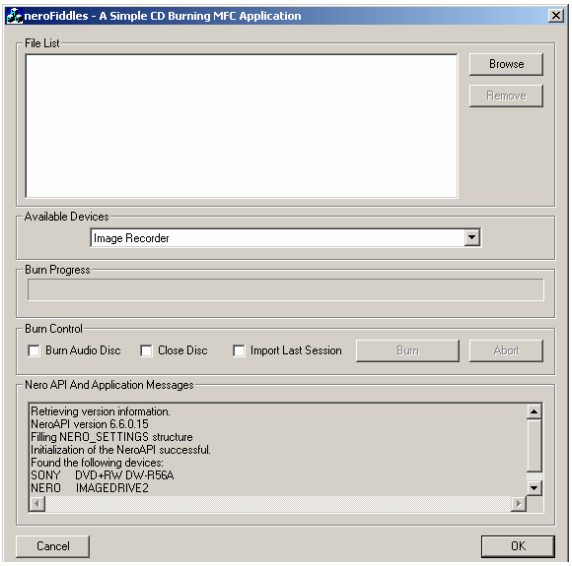
Line 1172: Here we can find the implementation of the various callback functions that the *NeroAPI* requires.

E. g. ProgressCallback, whenever called by the *NeroAPI*, will display the current progress of the burn progress in percent.

Line 1271: The NeroError function obtains the last error from the *NeroAPI*, prints some information about the action that failed, lets the *NeroAPI* free some memory and makes the application terminate with exit code 10.

6. Creating a Simple MFC Application

6.1. Nero Fiddled While Rome Burned!



It is quite obvious that the world has been waiting for an application that gives some tribute to important events of the past. So we will call this one “NeroFiddles”. Application names don’t come any better.

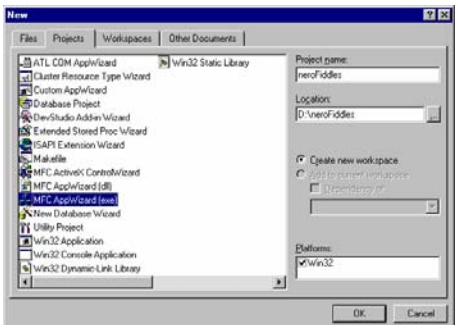
Our Nero - of course - has a lot more to do than just fiddle.

This simple application lists the available devices that can burn CDs. It lets the user choose a number of files and burns an ISO CD or Audio CD. This is nothing spectacular and less powerful than the command line examples in the previous chapter.

But to prevent the application from becoming too cluttered and hard to understand we have to keep it simple. GUI applications have a much bigger overhead than console applications. So we’ll just provide minimum functionality to keep the program small and simple. Once it works it is not that complicated to provide additional functionality. Getting started is the hard part.

6.2. Creating the Framework

This tutorial is based on Visual C++ 6.0. Visual Studio .NET screens may differ! Also Nero has to be installed on your system to run this sample!

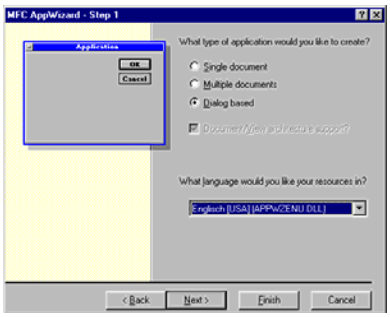


Open Microsoft Visual C++.

Select File/New from the menu. The “Projects” tab already should be selected when the “New” dialog opens.

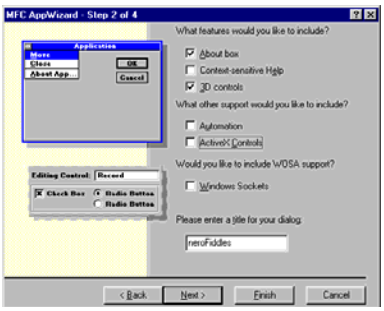
Select “MFC AppWizard (exe)” and type “NeroFiddles” into the “Project name” edit box. You may select your favorite project directory, but leave the rest of the settings as they are. Click on the “OK” button.

In the “MFC AppWizard Step 1” dialog select “Dialog based” and choose the preferred language for your resources. “English (USA) (APPWZENU.DLL)” should work fine, so let’s just pick this entry. Click “Next”.

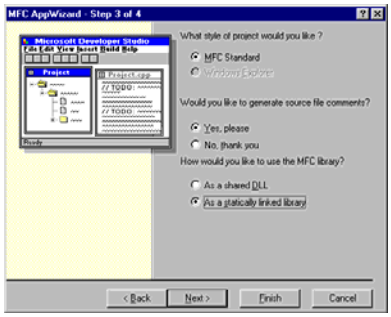


Uncheck “ActiveX Controls” in “MFC AppWizard Step 2” because we will not use any.

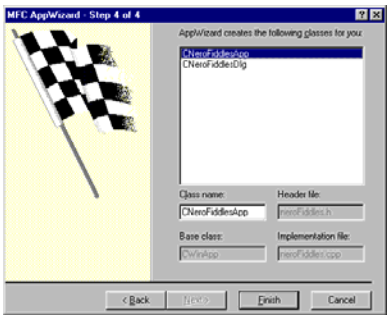
Click “Next”.



Select “As a statically linked library” in “MFC AppWizard Step 3 of 4”. The application becomes bigger, but we do not depend on the presence of the MFC DLLs on the target system. Leave the rest of the settings as they are. Click “Next”.

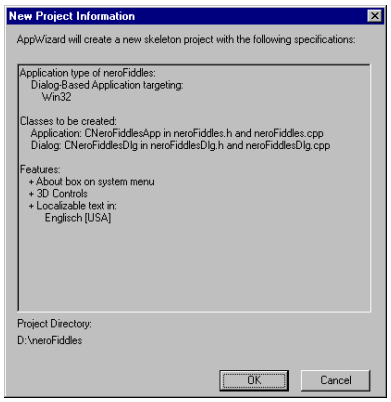


Click “Next”, do not make any changes and then click “Finish”.



A dialog pops up, telling you about what the AppWizard created for you. Click “OK”.

We now have a working MFC dialog based application that can be compiled and executed, though it doesn’t do much of what we need, yet.



6.3. Adding NeroAPI files

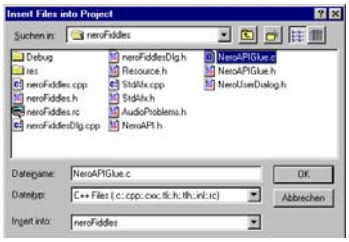
Make sure that you can see the workspace window on the left side. If it is not there, activate it by selecting View/Workspace from the menu.

Before we do anything else we need to make sure that everything we need from the *NeroAPI* is in its place. Go to your *NeroAPI* folder and copy the contents of the “include” and “lib” subdirectories to your NeroFiddles-project directory. You will need to copy the following files:

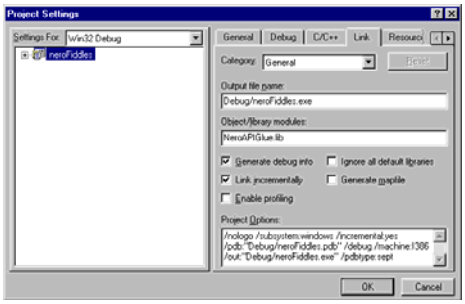
NeroAPI.h, NeroAPIGlue.h, NeroUserDialog.h and NeroAPIGlue.lib.

The required *NeroAPI* files now are located as desired. Return to Visual Studio.

Select Project/Add To Project/Files from the menu. Select “NeroAPIGlue.h” from the “Insert Files Into Project” dialog that comes up. Hold down the <Ctrl> key and also select “NeroAPI.h” and “NeroUserDialog.h”. Click “OK”.



Go to Project/Settings. Select the “Link” tab and pick “Input” in the “Category” dropdown list box. Type “NeroAPIGlue.lib” in the “Object/library modules” edit field and “libcmt.lib” in the “Ignore libraries” edit field.



Select the FileView tab in the Workspace window. Open “NeroFiddles files”/ “Header Files”/ “StdAfx.h” by double clicking it. Within that file you should find something like this:

```
#include <afxwin.h>         // MFC core and standard components
#include <afxext.h>         // MFC extensions
#include <afxdtctl.h>           // MFC support for Internet Explorer 4
Common Controls
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>             // MFC support for Windows Common Controls
#endif // _AFX_NO_AFXCMN_SUPPORT
```

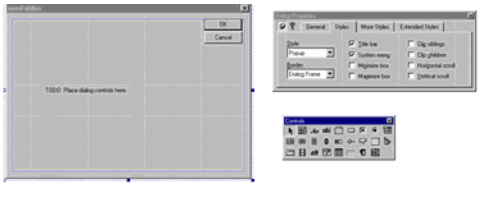
Right after that, add the following line:

```
#include "NeroAPIglue.h"
```

6.4. Adding Controls

Usually, the Resource View tab of the workspace window will already be selected and display a skeleton of our application. If you cannot see it, select the Resource View tab, and open NeroFiddles resources/Dialog/IDD_NEROFIDDLES_DIALOG.

You should see two buttons “OK”, “Cancel”, and a line of text that says “TODO: place your dialog controls here.”



That’s what we intended to do anyway. So click on that line of text, and delete it by pressing the “del”-key.

Resize the dialog window a little bit, so that it becomes bigger. There’s no need to squeeze everything into the small amount of space Visual Studio initially offers us.

Now click on the “List Box” icon in the little “Controls” window. You should place a List box control in the upper left corner of the NeroFiddles dialog. Resize the control so that it can display a little more text.

Right click over the List box and select “Properties” from the context menu that pops up. You will now see a “Edit properties” dialog. Click the “Keep Visible” pin-board-style pin icon in the upper left corner of the property dialog – we will need this dialog more than once.

Rename IDC_LIST1 to IDC_FILELIST.

We need a button that will later open a FileOpen dialog. Click on the button icon in the “Controls” window. Place the button to the right of the Edit control that you just inserted.

Change the name from “IDC_BUTTON1” to “IDC_BROWSE” in the “Push Button Properties” window. Change the “Caption” from “Button1” to “Browse”.

Add another button below the “Browse” button. Name it “IDC_REMOVE” and change the caption to “Remove”.

We also need a ComboBox that displays the available devices and lets the user select one for burning. Click on the “Combo Box” symbol of the “Control” window. Place the ComboBox under the Edit Box and resize it. Rename it from “IDC_COMBO1” to “IDC_DEVICES”. Select the “Styles” tab and change the type to “droplist” – the user then cannot enter any information, but has to choose from the options our application gives him, which is exactly what we want.

Now add a Progress Control and another button. Rename that button from “IDC_BUTTON3” to “IDC_BURN”. Change the caption to “Burn”. Make the button “disabled”. It will be enabled after the user has selected a file for burning.

Place another button to the right of IDC_BURN. Rename that button to IDC_ABORT. Change the caption to “Abort”. Make the button “disabled”; this is the initial state when the application starts. The Abort button will be enabled when the user pushes the Burn button.

Add an Edit Box; rename it from “IDC_EDIT1” to “IDC_MESSAGES”. Select the “Styles” tab and make it “multiline” and “read only”. Also check “Horizontal Scroll”, “AutoHScroll”, “Vertical Scroll” and “AutoVScroll”. Now resize the Edit Box so that it can display about ten lines of text.

Add three checkboxes and label them “Burn Audio Disc”, “Close Disc” and “Import Last Session”. Change the corresponding IDs to IDC_AUDIO, IDC_CLOSE and IDC_IMPORT.

That completes our work with the Resource Editor. You can enhance the appearance by using some group boxes if you want.

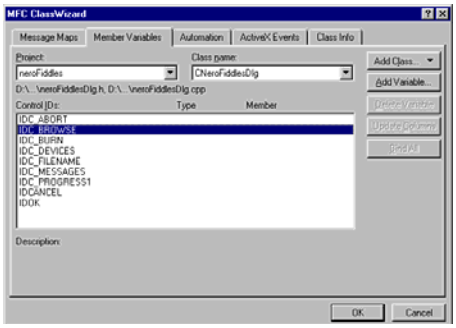
6.5. Adding Member Variables

If we build and run our application now, we see that it basically looks like what we wanted, but it doesn't do much so far. To include functionality we have to provide a few member variables, which map to controls and handle interchange with the *NeroAPI*.

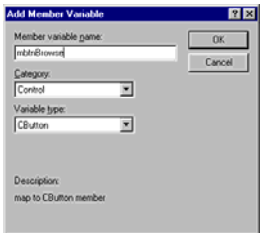
6.5.1. Variables for Controls

The controls we added need to be mapped to variables to provide easy access.

Open the ClassWizard (View/ClassWizard or Ctrl+W) and select the "Member Variables" tab. CNeroFiddlesDlg should already be selected as "Class name".



Click on "IDC_BROWSE" in the "Control IDs" list box. Click on "Add Variable", and in the following dialog provide the variable with the name "mbtnBrowse", make the Category "Control" and the variable type "CButton". Click "OK".



Now select "IDC_BURN", click on "Add Variable" and name it "mbtnBurn", category "Control", type "CButton".

IDC_ABORT gets a variable named "m_btnAbort", category Control, type CButton.

Here are the other mappings:

Control ID	Member variable	Variable Type
IDC_DEVICES	m_cbxDevices	CComboBox (category control; careful here: the default is value!),
IDC_FILELIST	m_lstFileList	CListBox
IDC_MESSAGES	m_edtMessages	CEdit
IDC_PROGRESS1	m_pgsProgress	CProgressCtrl
IDCANCEL	m_Cancel	CButton
IDOK	m_OK	CButton
IDC_CLOSE	m_chkbxClose	CButton
IDC_IMPORT	m_chkbxImport	CButton
IDC_AUDIO	m_chkbxAudio	CButton
IDC_REMOVE	m_btnRemove	CButton

The controls now have corresponding member variables and can be used quite easily.

6.5.2. Other Variables

We need to add numerous other variables to our dialog class.

Open the “ClassView” tab in the Workspace window. Right click on the “CNeroFiddlesDlg” class. Select “Add Member Variable” from the context menu. In the dialog that comes up enter “CString” for variable type and “m_strPathName” for variable name. Set “Access” to “private” and click “OK”.

Repeat this for CString mstrFileName, also private.

Enter “NERO_DEVICEHANDLE” as type and “m_ndhDeviceHandle” as variable name. Make it “private” and click on “OK”.

Repeat this with “NERO_SCSI_DEVICE_INFOS*” as type and “m_pndiDeviceInfos” as name. Make it “private”.

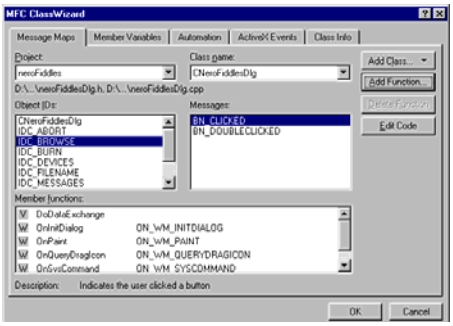
Do this for all of the following:

Variable Type	Variable
char	m_pcImage [256];
bool	m_bAborted
NERO_SETTINGS	m_nsSettings
char	m_pcLanguageFile [128]
char	m_pcSoftware [128]
char	m_pcVendor [128]
char	m_pcNeroFilesPath [128]
char	m_pcDriveName [128]
FILE*	m_pFile
NERO_ISO_ITEM*	m_pniiFile
NERO_WRITE_CD*	m_pnwcWriteCD
NERO_PROGRESS	m_npProgress
NERO_DEVICEHANDLE	m_ndhDeviceHandle
CString	m_strPathName

6.6. Adding Message Handling Functions For Controls

6.6.1. OnBrowse

The first “real” functionality we add is the selection of a file and the display of its name in IDC_FILENAME.



We need the ClassWizard again. If you closed it, reopen it and select the “Message Maps” tab. “ClassName” should still be CNeroFiddlesDlg. Select IDC_BROWSE from “Object IDs” and “BN_CLICKED” from “Messages”. Click “Add Function”.

Accept the proposed function name, which is “OnBrowse”, by clicking “OK”.

We have now added a message handler that calls the OnBrowse member function whenever the “Browse” button is clicked.

Click on “Edit Code”. The ClassWizard disappears and a source file window opens and displays the content of the OnBrowse function:

```
void CNeroFiddlesDlg::OnBrowse()
{
  // TODO: Add your control notification handler code here

}
```

Type the following after the line that starts with “TODO”. (To make things easier, you might as well copy it from here, if you obtained this document as a file.)

```
  static char BASED_CODE szFilterAudio[] = "MP3 Files (*.mp3)|*.mp3|All
Files (*.*)|*.*||";
  static char BASED_CODE szFilterData[] = "All Files (*.*)|*.*||";

  CFileDialog dlgOpen(TRUE, NULL, NULL, OFN_FILEMUSTEXIST |
OFN_ALLOWMULTISELECT, m_chkbxAudio.GetCheck() ? szFilterAudio :
szFilterData, this);

  if (dlgOpen.DoModal() == IDOK)
  {
      POSITION pos = dlgOpen.GetStartPosition();
      while (pos != NULL)
      {
            CString filename = dlgOpen.GetNextPathName(pos);
            if (m_lstFileList.FindString(0, filename) == LB_ERR)
```

```
            {
                  m_lstFileList.AddString(filename);
            }
    }
    if (m_pndiDeviceInfos->nsdisNumDevInfos > 0)
    {
      m_btnBurn.EnableWindow(true);
    }
  }
}
```

This code defines a control string for our allowed file types. If the Audio Disc checkbox is selected, preferrably MP3 files will be selected. If the disc is a data disc, all files are listed immediately. The string has to have a certain format so that it can be passed to the CFileDialog constructor. If the user clicks “OK” in the FileDialog, it will pass “IDOK” as return value.

Now all selected files are read one by one and added to the List box.

Afterwards, the functions checks whether any Devices have been enumerated during startup. If this is true, the Burn-button is enabled, otherwise it stays grayed.

6.6.2. OnRemove

When the “Remove” button is pressed, this function will empty the files names List box. This has to be done in reverse order.

```
void CNeroFiddlesDlg::OnRemove()
{
      int selCount = m_lstFileList.GetSelCount();
      if (selCount > 0)
      {
            int* selection = new int[selCount];
            m_lstFileList.GetSelItems(selCount, selection);
            for (int i = selCount - 1; i >= 0; --i)
            {
                  m_lstFileList.DeleteString(selection[i]);
            }
            delete[] selection;
      }
      m_btnRemove.EnableWindow(false);
}
```

6.6.3. OnSelchangeFilelist

When the file list selection changes, this function will either enable or disable the “Remove” button, based on the number of selected items.

```
void CNeroFiddlesDlg::OnSelchangeFilelist()
{
      int selCount = m_lstFileList.GetSelCount();
      m_btnRemove.EnableWindow(selCount > 0);
}
```

6.6.4. OnBurn

```
void CNeroFiddlesDlg::OnBurn()
{
```

First, the function checks whether or not the user had selected a file name by using the Browse button. If he did not, the function returns, doing nothing but adding an admonishing line to the message log.

```
  if (m_lstFileList.GetCount() == 0)
  {
    AppendString("You have to choose some files before
                                                you can start burning!");
  }
  else
  {
```

Get the number of audio files. The NERO_WRITE_CD struct already provides space for one audio track.

```
      int iNumAudio = 1;
      if(m_chkbxAudio.GetCheck())
      {
        for (int index = 0; index < m_lstFileList.GetCount(); ++index)
        {
          CString pathname;
          m_lstFileList.GetText(index, pathname);
          if(pathname.Right(4) == ".mp3")
                        iNumAudio++;
        }
      }
```

m_pnwcWriteCD should always be uninitialized at this place, but it doesn’t hurt to double-check.

```
      if(m_pnwcWriteCD != NULL)
         free(m_pnwcWriteCD), m_pnwcWriteCD = NULL;
```

Allocate the memory for all initialization data.

```
      size_t stTheSize  = sizeof(NERO_WRITE_CD) +
                        ((iNumAudio - 1) * sizeof(NERO_AUDIO_TRACK));
      m_pnwcWriteCD           = (NERO_WRITE_CD*)malloc(stTheSize);
```

```
      memset(m_pnwcWriteCD, 0, stTheSize);
```

Create NERO_ISO_ITEM struct from file list or even append audio tracks.

```
      NERO_ISO_ITEM* niiPrevItem = 0;
      int iTrackCnt                 = 0;
      for (int index = 0; index < m_lstFileList.GetCount(); ++index)
      {
        CString pathname;
        m_lstFileList.GetText(index, pathname);
        AppendString("Preparing data file:" + pathname);
        char path[MAX_PATH];
        char* name;
        GetFullPathName(pathname, MAX_PATH, path, &name);
         if(!m_chkbxAudio.GetCheck())
        {
           NERO_ISO_ITEM* pniiNewItem = NeroCreateIsoItem();
           memset(pniiNewItem, 0, sizeof(NERO_ISO_ITEM));
```

m_niiFile refers to the first item.

```
          if (m_pniiFile == 0)
                        m_pniiFile = pniiNewItem;
          pniiNewItem->longFileName      = strdup(name);
          pniiNewItem->longSourceFilePath= strdup(path);
          pniiNewItem->isDirectory       = FALSE;
          pniiNewItem->isReference       = FALSE;
          pniiNewItem->unicodeFileName   = NULL;
          pniiNewItem->nextItem          = NULL;
          if(niiPrevItem)
                        niiPrevItem->nextItem = pniiNewItem;
          niiPrevItem = pniiNewItem;
       }
       else if((pathname.Right(4) == ".mp3") && (iTrackCnt < iNumAudio))
       {
```

Initialize the audio tracks member of the NERO_WRITE_CD struct if we are about to write an Audio CD. Always add 2 seconds pause at the beginning of a track.

```
          m_pnwcWriteCD->nwcdTracks[iTrackCnt].
                                    natPauseInBlksBeforeThisTrack = 150;
```

The file has to be of MP3 kind.

```
                  m_pnwcWriteCD->nwcdTracks[iTrackCnt].
natSourceDataExchg.ndeType   = NERO_ET_FILE_MP3;
                  NERO_DATA_EXCHANGE& ndeCurrent                        =
m_pnwcWriteCD->nwcdTracks[iTrackCnt].natSourceDataExchg;
                  ndeCurrent.ndeData.ndeLongFileName.reserved    = 0;
                  ndeCurrent.ndeData.ndeLongFileName.ptr = strdup(path);
                  iTrackCnt++;
            }
      }
```

No CD stamp, artist or title are required.

```
    m_pnwcWriteCD->nwcdpCDStamp    = NULL;
```

```
    m_pnwcWriteCD->nwcdArtist      = NULL;
    m_pnwcWriteCD->nwcdTitle       = NULL;
```

No CD Extra information available.

```
    m_pnwcWriteCD->nwcdCDExtra           = FALSE;
```

Do we have Audio tracks?

```
    m_pnwcWriteCD->nwcdNumTracks   = iTrackCnt;
```

We want to write to a CD.

```
    m_pnwcWriteCD->nwcdMediaType   = MEDIA_CD;
```

Get the currently selected device from the ComboBox.

```
    int i = m_cbxDevices.GetCurSel();
```

Retrieve the NERO_SCSI_DEVICE_INFO pointer for the selected device and assign it to a local variable.

```
    NERO_SCSI_DEVICE_INFO* nsdiDevice =
                  (NERO_SCSI_DEVICE_INFO*)m_cbxDevices.GetItemDataPtr(i);
```

Try to open the selected device.

```
    m_ndhDeviceHandle = NeroOpenDevice(nsdiDevice);
```

Check whether a valid handle was returned.

```
    if (!m_ndhDeviceHandle)
    {
```

No handle available; tell the user what happened.

```
      AppendString("Device could not be opened: "+
                                  (CString)nsdiDevice->nsdiDeviceName);
    }
    else
    {
```

Now, import the last session if desired. If the function fails, there probably is no CD in the disc drive.

NeroImportDataTrack creates a NERO_ISO_ITEM tree from an already existing ISO track in order to create a new session with reference to files from older sessions.

m_pCDStamp will be filled with a pointer to a CDStamp object which has to be freed later by the CBurnContext destructor.

```
        if(m_chkbxImport.GetCheck())
        {
          NERO_IMPORT_DATA_TRACK_INFO nidtInfo;
          NERO_IMPORT_DATA_TRACK_RESULT nidtResult;
          CString csUserMsg         = "";
          memset(&nidtInfo, 0, sizeof(nidtInfo));
          nidtInfo.nidtiSize  = sizeof(nidtInfo);
          void* pCDStamp            = NULL;
          NERO_ISO_ITEM* pniiItem   = NULL;
          NERO_CD_INFO* pnciInfo    =
          NeroGetCDInfo(m_ndhDeviceHandle, 0);
```

```
        if(pnciInfo != NULL)
        {
           pniiItem = NeroImportDataTrack(m_ndhDeviceHandle,
                      pnciInfo->ncdiNumTracks - 1, &pCDStamp,
                      &nidtInfo, 0, &nidtResult, NULL);
           NeroFreeMem(pnciInfo);
        }
```

If there is a volume name after import, print it out.

```
        if(nidtInfo.nidtipVolumeName != NULL)
        {
          csUserMsg.Format("Imported volume name: %s",
                                      nidtInfo.nidtipVolumeName);
          AppendString(csUserMsg);
          NeroFreeMem (nidtInfo.nidtipVolumeName);
        }
```

If there was an error during import, let the user know about it.

```
        if((nidtResult != NIDTR_NO_ERROR) || (pniiItem == NULL))
        {
          static LPCSTR errors[] = {"an unknown error",
                                    "a generic error","a drive error",
                                    "a read error",
                                    "a filesystem error",
                                    "an invalid track number"};
           if (nidtResult > NIDTR_INVALID_TRACKNUMBER)
                 nidtResult = NIDTR_NO_ERROR;

           csUserMsg.Format("There was %s while importing the track!",
                                              errors[nidtResult]);
           AppendString(csUserMsg);
        }
        else if(m_pniiFile != NULL)
        {
```

Merge the new track with the existing one.

```
           if(MergeIsoTracks(&m_pniiFile, pniiItem))
                           m_pnwcWriteCD->nwcdpCDStamp = pCDStamp;
           else
           {
              AppendString("There was an error while merging tracks!");
              if(pCDStamp != NULL)
                      NeroFreeCDStamp(pCDStamp), pCDStamp = NULL;
           }
        }
     }
```

We have a valid device handle. While burning, the "Abort" button needs to be enabled. All the other buttons and controls have to be disabled.

```
     m_btnAbort.EnableWindow(true);
     m_Cancel.EnableWindow(false);
```

```
m_OK.EnableWindow(false);
m_cbxDevices.EnableWindow(false);
m_btnBrowse.EnableWindow(false);
m_btnBurn.EnableWindow(false);
m_btnRemove.EnableWindow(false);
m_chkbxAudio.EnableWindow(false);
m_chkbxImport.EnableWindow(false);
m_chkbxClose.EnableWindow(false);
```

Set the range for the progress control; the display unit will be percent.

```
m_pgsProgress.SetRange(0,100);
int iRes = NeroBurn(m_ndhDeviceHandle, NERO_ISO_AUDIO_CD,
            m_pnwcWriteCD, NBF_WRITE | (!m_chkbxClose.GetCheck() ?
            NBF_CLOSE_SESSION : 0), 0, &m_npProgress);
```

Free memory that was allocated for the track.

```
  if(!m_chkbxAudio.GetCheck() &&
                    (m_pnwcWriteCD->nwcdIsoTrack != NULL))
  NeroFreeIsoTrack(m_pnwcWriteCD->nwcdIsoTrack);
```

Close the device.

```
NeroCloseDevice(m_ndhDeviceHandle);
```

Burning is finished, disable "Abort" and activate all other controls.

```
m_btnAbort.EnableWindow(false);
m_Cancel.EnableWindow(true);
m_OK.EnableWindow(true);
m_cbxDevices.EnableWindow(true);
m_btnBrowse.EnableWindow(true);
m_btnBurn.EnableWindow(true);
m_btnRemove.EnableWindow(m_lstFileList.GetSelCount() > 0);
m_chkbxAudio.EnableWindow(!m_chkbxImport.GetCheck());
m_chkbxImport.EnableWindow(!m_chkbxAudio.GetCheck());
m_chkbxClose.EnableWindow(!m_chkbxAudio.GetCheck());
```

Clear the progress bar.

```
m_pgsProgress.SetPos(0);
```

Make sure that the aborted flag is not set if "Burn" button is pressed again.

```
m_bAborted = false;
```

Retrieve the error log.

```
char* Log = NeroGetErrorLog();
```

Display the error log contents

```
AppendString(Log);
```

Free the log

```
NeroFreeMem(Log);
```

Free the NERO_ISO_ITEM struct list.

```
DeleteIsoItemTree(m_pniiFile);
m_pniiFile = NULL;
```

Free the memory allocated for audio track file paths.

```
      for(int i = 0; i < iTrackCnt; i++)
      {
         const char* pcPath = m_pnwcWriteCD->
                              nwcdTracks[i].natSourceDataExchg.ndeData.
                              ndeLongFileName.ptr;
          if(pcPath != NULL)
          free((void*)pcPath), pcPath = NULL;
        }
```

Free the NERO_WRITE_CD struct, too.

```
        if(m_pnwcWriteCD != NULL)
        {
          if(m_pnwcWriteCD->nwcdpCDStamp != NULL)
          NeroFreeCDStamp(m_pnwcWriteCD->nwcdpCDStamp),
                                  m_pnwcWriteCD->nwcdpCDStamp = NULL;
          free(m_pnwcWriteCD), m_pnwcWriteCD = NULL;
        }
```

Tell the user how the burn process was finished.

```
      switch(iRes)
      {
        case NEROAPI_BURN_OK:
          AppendString ("BurnCD() : burn successful");
          break;
        case NEROAPI_BURN_UNKNOWN_CD_FORMAT:
          AppendString ("BurnCD() : unknown CD format");
          break;
        case NEROAPI_BURN_INVALID_DRIVE:
          AppendString ("BurnCD() : invalid drive");
          break;
        case NEROAPI_BURN_FAILED:
          AppendString ("BurnCD() : burn failed");
          break;
        case NEROAPI_BURN_FUNCTION_NOT_ALLOWED:
          AppendString ("BurnCD() : function not allowed");
          break;
        case NEROAPI_BURN_DRIVE_NOT_ALLOWED:
          AppendString ("BurnCD() : drive not allowed");
          break;
            case NEROAPI_BURN_USER_ABORT:
          AppendString ("BurnCD() : user aborted");
          break;
         case NEROAPI_BURN_BAD_MESSAGE_FILE:
          AppendString ("BurnCD() : bad message file");
          break;
        default:
          AppendString ("BurnCD() : unknown error");
              break;
```

```
      }
    }
  }
}
```

6.6.5. OnOK

When the application closes, we also need to properly disconnect from the NeroAPI.DLL. This means that we need to have handling-functions that intercept when the user clicks “OK” or “Cancel.

```
void CNeroFiddlesDlg::OnOK()
{
  NeroAPIFree();
  CDialog::OnOK();
}
```

6.6.6. OnCancel

Make the OnCancel handler to look like this:

```
void CNeroFiddlesDlg::OnCancel()
{
  // TODO: Add extra cleanup here
  NeroAPIFree();

  CDialog::OnCancel();
}
```

6.6.7. OnAbort

If the “Abort” button is pressed while burning, the member variable “mbAborted” will be set to true. The value of mbAborted will be returned to the *NeroAPI* during Process- and Idle-callbacks. If it becomes “true”, the *NeroAPI* will stop the burn process.

```
void CNeroFiddlesDlg::OnAbort()
{
  // TODO: Add your control notification handler code here
  m_bAborted = true;
}
```

6.6.8. OnAudio

If an audio disc shall be burnt, we cannot import previous sessions and the disc is always closed.

```
void CNeroFiddlesDlg::OnAudio()
{
  m_chkbxImport.EnableWindow(!m_chkbxAudio.GetCheck());
  m_chkbxClose.EnableWindow(!m_chkbxAudio.GetCheck());
  m_chkbxClose.SetCheck(m_chkbxAudio.GetCheck());
}
```

6.6.9. OnImport

Import is only possible for data discs.

```
void CNeroFiddlesDlg::OnImport()
{
  m_chkbxAudio.EnableWindow(!m_chkbxImport.GetCheck());
}
```

6.7. Adding Utility Functions

6.7.1. NeroAPIInit

The initialization of the *NeroAPI* will be performed during OnInitDialog. However, there is a lot to do, so we will not add the code there, but rather create a function that initializes the API.

This function will be named NeroAPIInit. The first thing we need to do is adding a line in OnInitDialog to calls this function.

Open the CNeroFiddlesDlg tree in ClassView and locate “OnInitDialog”. Double click “OnInitDialog”.

You will now see the body of this function. Go right to the end of it, where you should find some code that looks like this:

```
// Set the icon for this dialog.  The framework does this automatically
//  when the application's main window is not a dialog
SetIcon(m_hIcon, TRUE);            // Set big icon
SetIcon(m_hIcon, FALSE);           // Set small icon

// TODO: Add extra initialization here

return TRUE;  // return TRUE  unless you set the focus to a control
```

In the line after “TODO” type “NeroAPIInit;”. That will call the - still no-existing - function NeroAPIInit during initialization.

We will provide this function now.

Right click on CNeroFiddlesDlg and choose “Add Member Function”. Set the Function Type to “void” and the Function Declaration to “NeroAPIInit”. Click “OK”.

Visual Studio will add the function declaration, create the function body, and get you right into the function body. Fill the function with the following code:

```
m_bAborted = false;

AppendString("Opening NeroAPI.DLL");

if (!NeroAPIGlueConnect (NULL)) {
  AppendString("Cannot open NeroAPI.DLL");
  return;
}

AppendString("Retrieving version information.");

WORD majhi, majlo, minhi, minlo;

NeroGetAPIVersionEx(&majhi, &majlo, &minhi, &minlo, NULL);

CString strVersion;
strVersion.Format("Nero API version %d.%d.%d.%d",
                  majhi, majlo, minhi, minlo);

AppendString(strVersion);


AppendString("Filling NERO_SETTINGS structure");

strcpy(pcNeroFilesPath, "NeroFiles");
strcpy(pcVendor, "ahead");
strcpy(pcSoftware, "Nero - Burning Rom");
strcpy(pcLanguageFile, "Nero.txt");

memset(&nsSettings, 0, sizeof(nsSettings));
nsSettings.nstNeroFilesPath = pcNeroFilesPath;
nsSettings.nstVendor = pcVendor;
nsSettings.nstIdle.ncCallbackFunction = IdleCallback;
nsSettings.nstIdle.ncUserData = this;
nsSettings.nstSoftware = pcSoftware;
nsSettings.nstUserDialog.ncCallbackFunction = UserDialog;
nsSettings.nstUserDialog.ncUserData = this;
nsSettings.nstLanguageFile =pcLanguageFile;

memset(&npProgress, 0, sizeof(npProgress));
npProgress.npAbortedCallback = AbortedCallback;
npProgress.npAddLogLineCallback = AddLogLine;
npProgress.npDisableAbortCallback = NULL;
npProgress.npProgressCallback = ProgressCallback;
```

```
  npProgress.npSetPhaseCallback = SetPhaseCallback;
  npProgress.npSetMajorPhaseCallback=NULL;
  npProgress.npSubTaskProgressCallback=NULL;
  npProgress.npUserData = this;

  pndiDeviceInfos = NULL;

  NEROAPI_INIT_ERROR initErr;
  initErr = NeroInit (&nsSettings, NULL);

  switch (initErr)
  {
    case NEROAPI_INIT_OK:
      AppendString("Initialization of the NeroAPI successful.");
      break;
    case NEROAPI_INIT_INVALID_ARGS:
      AppendString("The arguments are not valid.");
      break;
    case NEROAPI_INIT_INVALID_SERIAL_NUM:
      AppendString("The Serial Number is not valid.");
      break;
    default:
      AppendString("An error occured. The type of error cannot be
determined.");
      break;
  }

  pndiDeviceInfos = NeroGetAvailableDrivesEx (MEDIA_CD, NULL);

  if (!pndiDeviceInfos) {
    AppendString("NeroGetAvailableDrives() returned no available
devices.");
  }
  else
  {
    if (pndiDeviceInfos->nsdisNumDevInfos > 0)
    {
      AppendString("Found the following devices:");
      for (DWORD dDeviceCounter = 0; dDeviceCounter <
                 pndiDeviceInfos->nsdisNumDevInfos; dDeviceCounter++)
      {
        AppendString(pndiDeviceInfos->
                       nsdisDevInfos[dDeviceCounter].nsdiDeviceName);

        int i = mcbxDevices.AddString(pndiDeviceInfos->
                       nsdisDevInfos[dDeviceCounter].nsdiDeviceName);

        mcbxDevices.SetItemDataPtr(i, &pndiDeviceInfos->
                                         nsdisDevInfos[dDeviceCounter]);
```

```
      }
      mcbxDevices.SelectString(-1, pndiDeviceInfos-
>nsdisDevInfos[0].nsdiDeviceName);
    }
    else
    {
      AppendString("The number of available devices is 0.");
    }
  }
```

The NERO_SETTINGS and NERO_PROGRESS structures are initialized and then filled with pointers to callback functions, as well as the this-pointer.

The result of the call to NeroInit is evaluated and added to the message log.

Then the available drives are added to the “Devices” ComboBox, linking each entry with a pointer to a NERO_SCSI_DEVICE_INFO.

6.7.2. NeroAPIFree

This function disconnects NeroFiddles from the *NeroAPI* and is called when our application closes.

Right click on CNeroFiddlesDlg and choose “Add Member Function”. Set the Function Type to “void” and the Function Declaration to “NeroAPIFree”. Click “OK”.

Visual Studio will add the function declaration, create the function body and set the cursor to the function body. Fill the function with the following code:

```
void CNeroFiddlesDlg::NeroAPIFree()
{
  if (m_pndiDeviceInfos)
  {
    NeroFreeMem(m_pndiDeviceInfos);
  }

  NeroClearErrors();
  if(NeroDone())
  {
        AfxMessageBox("Detected memory leaks in NeroFiddles");
  }

  NeroAPIGlueDone();

  return;
}
```

6.7.3. AppendString

Right click on CNeroFiddlesDlg and choose “Add Member Function”. Set the Function Type to “void” and the Function Declaration to “AppendString (CString str)”. Click “OK”.

Fill the function with the following code:

```
CString   strBuffer;

m_edtMessages.GetWindowText (strBuffer);
if (!strBuffer.IsEmpty())
{
   strBuffer += "\r\n";
}
strBuffer += str;
m_edtMessages.SetWindowText (strBuffer);

m_edtMessages.LineScroll (m_edtMessages.GetLineCount(), 0);
```

6.7.4. GetFilename

This function returns the correct file name of a NERO_ISO_ITEM.

```
inline LPCSTR GetFilename(const NERO_ISO_ITEM* pItem)
{
      return (pItem->longFileName != NULL)? pItem->longFileName : pItem-
>fileName;
}
```

6.7.5. MergeIsoTracks

The following function performs a merge operation between two iso item trees. The second tree is added onto the first one and the extra items are deleted.

As we do not allow adding directories in our file list, we do not make recursive calls of this method.

```
BOOL CNeroFiddlesDlg::MergeIsoTracks(NERO_ISO_ITEM** ppniiTarget,
NERO_ISO_ITEM* pniiToAdd)
{
  BOOL bResult = ((ppniiTarget != NULL) && (*ppniiTarget != NULL) &&
                                              (pniiToAdd != NULL));
```

There are wwo loops. The outer loop corresponds to the first tree, the inner loop to the second tree.

```
  for( ; bResult && (*ppniiTarget != NULL);
                              ppniiTarget = &(*ppniiTarget)->nextItem)
  {
    for(NERO_ISO_ITEM** ppniiToAddLocal = &pniiToAdd;
                                         *ppniiToAddLocal != NULL; )
```

```
    {
```

Compare entry names...

```
      if(0 == stricmp(GetFilename(*ppniiTarget),
                                        GetFilename(*ppniiToAddLocal)))
      {
```

If there is a file name conflict between ISO items that belong to imported sessions, always replace the old files according to the modification timestamp.

```
        time_t timeTarget = mktime(&(*ppniiTarget)->entryTime);
        if(timeTarget == (time_t)-1)
        {
          HANDLE hFile = NULL;
          FILETIME ftCreationTime;
          FILETIME ftLastAccessTime;
          FILETIME ftLastWriteTime;
```

Open the file, to get a handle.

```
          hFile = CreateFile((*ppniiTarget)->longSourceFilePath,
                                          GENERIC_READ,
                                          FILE_SHARE_READ,
                                          NULL,
                                          OPEN_EXISTING,
                                          FILE_ATTRIBUTE_NORMAL,
                                          NULL);
          if(hFile == INVALID_HANDLE_VALUE)
          {
            CString csMsg;
            csMsg.Format("Could not open file: %s.",
                                  (*ppniiTarget)->longSourceFilePath);
            AppendString(csMsg);
          }
          else if(GetFileTime(hFile, &ftCreationTime,
                                  &ftLastAccessTime, &ftLastWriteTime))
                       timeTarget = CTime(ftLastWriteTime).GetTime();

          if(hFile != INVALID_HANDLE_VALUE)
                       CloseHandle(hFile), hFile = NULL;
        }

        time_t timeToAdd = mktime(&(*ppniiToAddLocal)->entryTime);
```

If we have to replace one item, we will now switch places of items in the first and second tree. Since one of the items has to be deleted eventually, this operation will essentially keep the item in the second tree and delete the item from the first tree.

```
        if(timeTarget < timeToAdd)
        {
          NERO_ISO_ITEM* pniiTmpItem     = *ppniiToAddLocal;
          *ppniiToAddLocal               = *ppniiTarget;
          *ppniiTarget                   = pniiTmpItem;
```

```
          pniiTmpItem                    = (*ppniiToAddLocal)->nextItem;
          (*ppniiToAddLocal)->nextItem   = (*ppniiTarget)->nextItem;
          (*ppniiTarget)->nextItem       = pniiTmpItem;
        }
```

Remove the item from the second tree.

```
        NERO_ISO_ITEM* pniiTmpItem       = *ppniiToAddLocal;
        *ppniiToAddLocal                 = pniiTmpItem->nextItem;
        pniiTmpItem->nextItem      = NULL;
        DeleteIsoItemTree(pniiTmpItem);
      }
```

No match, advance to the next item.

```
      else
          pniiToAddLocal = &(*ppniiToAddLocal)->nextItem;
    }
  }
```

Attach whatever is left of the new tree to the main tree.

```
  *ppniiTarget = pniiToAdd;
```

Returning true means, everything is fine, continue.

```
      return bResult;
}
```

6.7.6. DeleteIsoItemTree

This function deletes the ISO tree recursively.

```
void CNeroFiddlesDlg::DeleteIsoItemTree(NERO_ISO_ITEM* pniiItem)
{
```

First free our own long filename strings, then free the whole tree.

```
      FreeOurOwnResources(pniiItem);
      NeroFreeIsoItemTree(pniiItem);
}
```

6.7.7. FreeOurOwnResources

Free resources allocated by NeroFiddles.

```
void CNeroFiddlesDlg::FreeOurOwnResources(NERO_ISO_ITEM* pniiItem)
{
```

Step through the tree until the ISO item tree pointer becomes NULL

```
  while(pniiItem != NULL)
  {
    NERO_ISO_ITEM* pniiNextItem = pniiItem->nextItem;
```

We have encountered another ISO item tree; recurse another level.

```
    if(pniiItem->isDirectory)
                  FreeOurOwnResources (pniiItem->subDirFirstItem);
    if(!pniiItem->isReference)
```

```
    {
      if(pniiItem->longFileName != NULL)
                        free((void*)pniiItem->longFileName),
                                        pniiItem->longFileName = NULL;
      if(pniiItem->longSourceFilePath != NULL)
                        free((void*)pniiItem->longSourceFilePath),
                                    pniiItem->longSourceFilePath = NULL;
    }
    pniiItem = pniiNextItem;
  }
}
```

6.8. Adding Callback Functions

The one remarkable thing about the use of the callback functions is the this-pointer.

The ncUserData member of the NERO_CALLBACK structure is supposed to hold a pointer to the calling object in a C++ environment. We filled that pointer with a dummy value in the console applications, because there is no such pointer when you are not using classes and objects.

For NeroFiddles it is vital, though. If we do not hand over that pointer to the *NeroAPI* and retrieve it in our callback functions, we will not be able to access any non-static member of our CNeroFiddlesDlg class. This would mean that we could not update the progress bar or print messages, which is a must.

We set the this-pointer during NeroAPIInit:

```
nsSettings.nstUserDialog.ncUserData = this;
```

The *NeroAPI* stores the pointer, and what we need to do is retrieve it. It is handed over to our callback functions as void* pUserData. We have to cast it to a CNeroFiddlesDlg pointer. The usage looks like this:

```
bool bSomeBooleanVariable = ((CNeroFiddlesDlg*)pUserData)->mbAborted;
```

6.8.1. IdleCallback

IdleCallback will be called continuously during a burn process. If the user clicked the “Abort” button, mbAborted becomes true, and the API will be told to stop burning.

Use the known ClassView approach to add a member function. Set the Function Type to “BOOL NERO_CALLBACK_ATTR” and the Function Declaration to “IdleCallback (void *pUserData)”. Activate the “static” Checkbox.

Now change the function body to this:

```
BOOL NERO_CALLBACK_ATTR CNeroFiddlesDlg::IdleCallback(void *pUserData)
{
  static MSG msg;
  while (!(((CNeroFiddlesDlg*)pUserData)->m_bAborted) &&
                        ::PeekMessage(&msg,NULL,NULL,NULL,PM_NOREMOVE))
  {
    if (!AfxGetThread()->PumpMessage())
    {
      break;
    }
  }
  return ((CNeroFiddlesDlg*)pUserData)->m_bAborted;
}
```

The first part of the function ensures that, while Nero is burning, the application still can process messages.

6.8.2. UserDialog

The UserDialog callback function is designed to let the user make a choice, or tell the *NeroAPI* that the user had finished a task, which the *NeroAPI* required him to perform. To keep the sample as small as possible, we will only provide user-controlled handling where absolutely required.

Add a member function of Function Type “NeroUserDlgInOut NERO_CALLBACK_ATTR” and make the Function Declaration “UserDialog (void *pUserData, NeroUserDlgInOut type, void *data)”. Activate the “static” Checkbox.

Make the function body look like this:

```
NeroUserDlgInOut NERO_CALLBACK_ATTR CNeroFiddlesDlg::UserDialog(void
*pUserData, NeroUserDlgInOut type, void *data)
{
  switch (type)
  {
    case DLG_AUTO_INSERT:
      return DLG_RETURN_CONTINUE;
      break;
    case DLG_DISCONNECT_RESTART:
      return DLG_RETURN_ON_RESTART;
      break;
    case DLG_DISCONNECT:
      return DLG_RETURN_CONTINUE;
      break;
    case DLG_AUTO_INSERT_RESTART:
      return DLG_RETURN_EXIT;
      break;
    case DLG_RESTART:
```

```
      return DLG_RETURN_EXIT;
      break;
    case DLG_SETTINGS_RESTART:
      return DLG_RETURN_CONTINUE;
      break;
    case DLG_OVERBURN:
      return DLG_RETURN_TRUE;
      break;
    case DLG_AUDIO_PROBLEMS:
      return DLG_RETURN_EXIT;
      break;
    case DLG_FILESEL_IMAGE:
      {
        static char BASED_CODE szFilter[] = "Image Files
                              (*.nrg)|*.nrg|All Files (*.*)|*.*||";

        CFileDialog dlgOpen(TRUE, NULL, "test.nrg", OFN_OVERWRITEPROMPT,
                              szFilter, ((CNeroFiddlesDlg*)pUserData));

        if (dlgOpen.DoModal() == IDOK)
        {
          strcpy((char*)data,dlgOpen.GetPathName());
          return DLG_RETURN_TRUE;
        }
        else
        {
          return DLG_BURNIMAGE_CANCEL;
        }
      }
      break;
    case DLG_WAITCD:
    {
      NERO_WAITCD_TYPE waitcdType = (NERO_WAITCD_TYPE) (int)data;
        char *waitcdString = NeroGetLocalizedWaitCDTexts (waitcdType);
      ((CNeroFiddlesDlg*)pUserData)->AppendString(waitcdString);
        NeroFreeMem(waitcdString);
      return DLG_RETURN_EXIT;
      break;
    }
    default:
      break;
  }
  return DLG_RETURN_EXIT;
}
```

6.8.3. ProgressCallback

The ProgressCallback function will provide information on how much of the current process has been completed. We use this information for display in a progress bar.

Add a member function of Function Type “BOOL NERO_CALLBACK_ATTR” and make the Function Declaration “ProgressCallback (void *pUserData, DWORD dwProgressInPercent)”. Activate the “static” Checkbox.

Make the function body look like this:

```
BOOL NERO_CALLBACK_ATTR CNeroFiddlesDlg::ProgressCallback(void
*pUserData, DWORD dwProgressInPercent)
{
  ((CNeroFiddlesDlg*)pUserData)->
                              m_pgsProgress.SetPos(dwProgressInPercent);
  return ((CNeroFiddlesDlg*)pUserData)->mbAborted;
}
```

6.8.4. AbortedCallback

This function is used by the *NeroAPI* to check whether the current process is supposed to be terminated.

Add a member function of Function Type “BOOL NERO_CALLBACK_ATTR” and make the Function Declaration “AbortedCallback(void *pUserData)”. Activate the “static” Checkbox.

Make the function body look like this:

```
BOOL NERO_CALLBACK_ATTR CNeroFiddlesDlg::AbortedCallback(void*
                                                                pUserData)
{
  return ((CNeroFiddlesDlg*)pUserData)->m_bAborted;
}
```

6.8.5. AddLogLine

This function provides textual information about certain states that might be important for the application.

Add a member function of Function Type “void NERO_CALLBACK_ATTR” and make the Function Declaration “AddLogLine(void *pUserData, NERO_TEXT_TYPE type, const char *text)”.

Activate the “static” Checkbox.

Make the function body look like this:

```
void NERO_CALLBACK_ATTR CNeroFiddlesDlg::AddLogLine(void *pUserData,
NERO_TEXT_TYPE type, const char *text)
{
  CString csTemp(text);
  ((CNeroFiddlesDlg*)pUserData)->AppendString("Log line:" + csTemp);
  return;
}
```

6.8.6. SetPhaseCallback

This function provides textual information about the current phase of the burning process.

Add a member function of Function Type “void NERO_CALLBACK_ATTR ” and make the Function Declaration “ SetPhaseCallback(void *pUserData, const char *text)”. Activate the “static” Checkbox.

Make the function body look like this:

```
void NERO_CALLBACK_ATTR CNeroFiddlesDlg::SetPhaseCallback(void
*pUserData, const char *text)
{
  CString csTemp(text);
  ((CNeroFiddlesDlg*)pUserData)->AppendString("Phase: " + csTemp);
  return;
}
```

6.9. Build and Run NeroFiddles

We are almost done. We have added everything that is required; now choose “Build/Rebuild All” from the menu and then “Build/Execute NeroFiddles.exe”.

If we did everything right, NeroFiddles should now be running.

NeroFiddles is almost screaming for additional functionality. You should check the command line examples, and get ideas there. For example, you could enable it to burn more than one file or complete folders. You could complete the user interaction part and provide burning of different formats.

You could provide RadioButtons to toggle between simulation of the burn process and actual burning. Also, you could add the “continue session” feature.

7. API Types and Functions

This paragraph describes the interface to the *NeroAPI* DLL.

7.1. Types

7.1.1. DLG_OVERBURN_INFO

Additional information when DLG_OVERBURN user callback is called.

```
typedef struct
{
      DWORD dwTotalBlocksOnCD;
      DWORD dwTotalCapacity;
      DWORD reserved[32];
} DLG_OVERBURN_INFO;
```

Description of structure members	
dwTotalBlocksOnCD	Total blocks to be written to disc.
dwTotalCapacity	Free capacity on disc in blocks.
reserved	Reserved for future usage.

Identifier	Introduced in *NeroAPI* version
DLG_OVERBURN_INFO	6.0.0.27

7.1.2. NERO_ABORTED_CALLBACK

TRUE indicates that the user wants to abort.

```
typedef BOOL  (NERO_CALLBACK_ATTR *NERO_ABORTED_CALLBACK)
(void *pUserData);
```

Identifier	Introduced in *NeroAPI* version
NERO_ABORTED_CALLBACK	5.0.3.9

7.1.3. NERO_ACCESSTYPE

This type is used when querying the available speeds for reading or writing with the NeroGetAvailableSpeeds function.

```
typedef enum
{
   ACCESSTYPE_WRITE,
   ACCESSTYPE_READ
} NERO_ACCESSTYPE;
```

Identifier	Introduced in *NeroAPI* version
NERO_ACCESSTYPE	5.5.9.14

7.1.4. NERO_ADD_LOG_LINE_CALLBACK

A one-line text is to be displayed. The text pointer becomes invalid after returning from this function.

```
typedef void  (NERO_CALLBACK_ATTR *NERO_ADD_LOG_LINE_CALLBACK)
(void *pUserData, NERO_TEXT_TYPE type, const char *text);
```

Identifier	Introduced in *NeroAPI* version
NERO_ADD_LOG_LINE_CALLBACK	5.0.3.9

7.1.5. NERO_AUDIO_FORMAT_INFO

A pointer to a variable of this type is returned by the NeroAudioGetFormatInfo function.

```
typedef struct tagNERO_AUDIO_FORMAT_INFO
{
      char  nafiDescription[256],
            nafiExtList[256];
      BOOL  nafiTgt,
            nafiConfigurable;
}     NERO_AUDIO_FORMAT_INFO;
```

Description of structure members	
nafiDescription[256]	A description, e.g. "RIFF PCM WAV format".
nafiExtList[256];	A list, e.g. "wav,wave,riff"
nafiTgt	Contains TRUE if this is a target plug-in.
nafiConfigurable	ConfigureItem will fail on items of this type if this member equals to false.

Identifier	Introduced in *NeroAPI* version
NERO_AUDIO_FORMAT_INFO	5.5.9.8

7.1.6. NERO_AUDIO_ITEM_INFO

This type is used as a member of the NERO_DATA_EXCHANGE struct.

```
typedef struct tagNERO_AUDIO_ITEM_INFO
{
  NERO_AUDIO_ITEM_HANDLE     naiiAudioItem;
  const char                 *naiiFileName;
} NERO_AUDIO_ITEM_INFO;
```

Identifier	Introduced in *NeroAPI* version
NERO_AUDIO_ITEM_INFO	5.5.9.14

7.1.7. NERO_AUDIO_ITEM_HANDLE

This handle is returned by the NeroAudioCreateTargetItem helper function. NERO_AUDIO_ITEM_INFO contains a NERO_AUDIO_ITEM_HANDLE member.

```
typedef void * NERO_AUDIO_ITEM_HANDLE;
```

Identifier	Introduced in *NeroAPI* version
NERO_AUDIO_ITEM_HANDLE	5.5.9.14

7.1.8. NERO_AUDIO_TRACK

NERO_AUDIO_TRACK is used as member of the NERO_WRITE_CD struct.

```
typedef struct tag_NERO_AUDIO_TRACK
{
      DWORD natPauseInBlksBeforeThisTrack;
      DWORD natNumIndexPositions;
      DWORD natRelativeIndexBlkPositions[98];
      const char *natTitle, *natArtist;
      NERO_DATA_EXCHANGE natSourceDataExchg;
      DWORD natLengthInBlocks;
      BOOL  natIndex0ContainsData;
      DWORD natReserved[31];
} NERO_AUDIO_TRACK;
```

Description of structure members	
natPauseInBlksBeforeThisTrack	Pause in blocks before this track.
natNumIndexPositions	Number of index positions.
natRelativeIndexBlkPositions	Offsets between one index position and the next one. **Note:** The values of index positions have to be given in bytes, whereby the values have to be a multiple of 2352.
natTitle	Set to NULL if unknown or to be taken from source.
natArtist	Set to NULL if unknown or to be taken from source.
natSourceDataExchg	Contains information about the type of data exchange (file,

Description of structure members	
	callback, audio item).
natLengthInBlocks	Only used for NERO_IO_CALLBACK.
natReserved	Should be zero.
natIndex0ContainsData	TRUE, if audio data shall be written into index 0. Data for index 0 must be provided. This can be used to prevent silent pauses between tracks.

Identifier	Introduced in *NeroAPI* version
NERO_AUDIO_TRACK	5.0.3.9
natIndex0ContainsData	5.5.9.8
natReserved	5.5.9.8: Size decreased from 32 to 31.

7.1.9. NERO_CALLBACK

Actually, this is a pointer to one of several different callback functions defined below. ncUserdata will be passed to the function as first parameter when it is called by the *NeroAPI*.

A callback function is an interface to other software to notify your application of changes. Windows makes extensive use of callback functions.

Data exchange between an application and *NeroAPI* is done with a function that gets a pointer to its own structure, a buffer pointer, and the amount of bytes to be read or written. It shall return the actual amount of bytes transferred. Other functions indicate that the end of the file has been reached (EOF) when reading, or that a serious error occurred.

```
typedef struct tag_NERO_CALLBACK
{
  void *ncCallbackFunction;
  void *ncUserData
} NERO_CALLBACK;
```

Identifier	Introduced in *NeroAPI* version
NERO_CALLBACK	5.0.3.9

7.1.10. NERO_CD_COPY

To copy a disc with the *NeroAPI*, fill in the NERO_CD_COPY structure and pass it to the NeroBurn function as parameter pWriteCD. Also, set the CDFormat parameter of NeroBurn to NERO_CD_COPY_TYPE.

```
#define NERO_CD_COPY_TYPE    ((NERO_CD_FORMAT)1000)
```

```
typedef struct
{
      NERO_DEVICEHANDLE sourceDrive;
      BOOL onTheFly;
      const char *imageFilePath;
      BOOL deleteImage;

      int readSpeed;

      UINT tryNr;
      BOOL readErrOption;
      BOOL readRawMode1;
      int rawMode1Option;
      BOOL readRawMode2;
      int rawMode2Option;

      BOOL readSub;
      BOOL checkDA;
      int slowDown;
      BOOL ignoreDAErr;
      BOOL readIsrcAndMediaCatalogNo;
      BOOL ignoreBadTOCType;

      NERO_MEDIA_TYPE mediaType;
} NERO_CD_COPY;
```

Description of structure members	
sourceDrive	The drive to read the data from.
onTheFly	Copy on-the-fly (without storing an image on the hard drive first).
imageFilePath	For non on-the-fly copying, specify the temporary image file path (mandatory).
deleteImage	Determine whether or not the temporary image shall be deleted after burning.
readSpeed	Read speed in KB/s, 0 for maximum speed.
tryNr	Determine the number of read attempts if a read error occurs.
readErrOption	For data tracks: 1 - ignore read errors and continue. 0 - abort on read errors.

Description of structure members	
	For *NeroAPI* 5.5 it was possible to select the readRaw mode for Mode1 and Mode2 separately. Up from *NeroAPI* 6, general read raw mode is enabled if one of these options is set.
readRawMode1	Read raw mode.
rawMode1Option	For *NeroAPI* 6: ignored; For *NeroAPI* 5.5: 0 - Repair EDC/ECC error for raw data. 1 - Ignore EDC/ECC error for raw data.
readRawMode2	Read raw mode.
rawMode2Option	For *NeroAPI* 6: ignored; For *NeroAPI* 5.5: 0 - Repair EDC/ECC error for raw data. 1 - Ignore EDC/ECC error for raw data.
readSub	For *NeroAPI* 6: ignored. For *NeroAPI* 5.5: read audio data with sub q code.
checkDA	For *NeroAPI* 6: ignored. For *NeroAPI* 5.5: check for audio data.
slowDown	For *NeroAPI* 6: ignored. For *NeroAPI* 5.5 option for jitter correction in read audio track: 0 - If correction failed rewind and read from the beginning. 1 - If correction failed slow down at once. 2 - No jitter correction.
ignoreDAErr	For audio tracks: Ignore read errors and continue.
readIsrcAndMediaCatalogNo	*NeroAPI* 5.5 and *NeroAPI* after 6.3.1.24: TRUE if media catalog number and ISRC should be read and copied.
ignoreBadTOCType	For *NeroAPI* 6: ignored. For *NeroAPI* 5.5: should be set to TRUE.
mediaType	*NeroAPI* 6.3.1.24 and later: Determine which media type is to be copied.

Identifier	Introduced in *NeroAPI* version
NERO_CD_COPY	6.3.1.24

7.1.11. NERO_CD_FORMAT

Used in the NeroBurn function to determine the format that will be written on the media.

Due to historical reasons, the enum type and some of its enumerators refer to CD media. Please bear in mind that this type can be used with DVD media as well!

```
typedef enum
{
  NERO_ISO_AUDIO_MEDIA             =0,
  NERO_VIDEO_CD                    =1,
  NERO_BURN_IMAGE_MEDIA            =2,
  NERO_FREESTYLE_CD                =3,
  NERO_FILE_SYSTEM_CONTAINER_MEDIA =4,

  NERO_ISO_AUDIO_CD                =0,
  NERO_BURN_IMAGE_CD               =2
} NERO_CD_FORMAT;
```

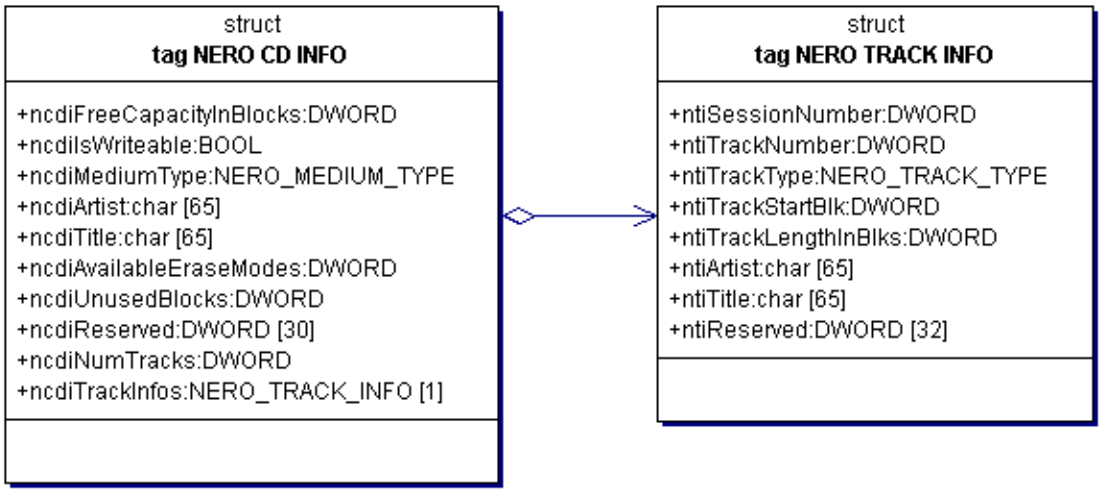
Description of enumerators	
NERO_ISO_AUDIO_MEDIA	Burn either a CD or a DVD, depending on the nwcdMediaType member.
NERO_VIDEO_CD	
NERO_BURN_IMAGE_MEDIA	Burn either a CD or a DVD from an image.
NERO_FREESTYLE_CD	For a Freestyle compilation.
NERO_FILE_SYSTEM_ CONTAINER_MEDIA	Burn an IfileSystemDescContainer.
NERO_ISO_AUDIO_CD	Audio or ISO CD. Available only for compatibility reasons.
NERO_BURN_IMAGE_CD	CD Type determined by content of CD Image. Available only for compatibility reasons.

Identifier	Introduced in *NeroAPI* version
NERO_CD_FORMAT	5.0.3.9
NERO_FILE_SYSTEM_CONTAINER_MEDIA	5.5.6.0

7.1.12. NERO_CD_INFO

This type is returned by the NeroGetCDInfo function and provides detailed information about the current media.

Due to historical reasons, this type refers to CD media. Please bear in mind that this type can be used with DVD media as well!




```
typedef struct tag_NERO_CD_INFO
{
      DWORD               ncdiFreeCapacityInBlocks;
      BOOL                ncdiIsWriteable;
      NERO_MEDIUM_TYPE    ncdiMediumType;
      char                ncdiArtist[65];
      char                ncdiTitle[65];
      DWORD               ncdiAvailableEraseModes;
      DWORD               ncdiUnusedBlocks;
      NERO_MEDIA_TYPE     ncdiMediaType;
      DWORD               ncdiMediumFlags;
      DWORD               ncdiLayer0MaxBlocks;
      DWORD               ncdiTotalCapacity;
      DWORD               ncdiReserved[25];
      NERO_BOOK_TYPE      ncdiBookType;
      DWORD               ncdiNumTracks;
      NERO_TRACK_INFO     ncdiTrackInfos[1];
} NERO_CD_INFO;
```

Description of structure members	
ncdiFreeCapacityInBlocks	Number of unused blocks on CD.
ncdiIsWriteable	A disc can be non-writeable.
ncdiMediumType	Old media type description, ncdiMediaType should be used instead.

Description of structure members		
ncdiArtist	Artist name.	
ncdiTitle	CD Title.	
ncdiAvailableEraseModes	This bitfield can be decoded using the NCDI_IS_ERASE_MODE_AVAILABLE macro.	
ncdiUnusedBlocks	Difference between Lead-Out position and last possible Lead-Out position.	
ncdiMediaType	Type of media.	
ncdiMediumFlags	Various media flags:	
	NCDIMF_ VIRTUALMULTISESSION	The media is a virtual multisession media; use VMS API to retrieve session information. Note: This flag only tells you, that if multisession is written, VMS is used. But not that this medium contains multisessions.
	NCDIMF_HDB_SUPPORTED	The media supports HD-BURN.
	NCDIMF_FINALIZED_DISC	Disc is finalized. No further writing possible.
ncdiLayer0MaxBlocks	If this value is set, the medium is a double layer medium whereby layer 0 can not be bigger than the given number of blocks.	
ncdiTotalCapacity	The total capacity of this medium.	
ncdiBookType	Book type of current disc or NBT_UNKNOWN, if it could not be read.	
ncdiReserved	Should be zero.	
ncdiNumTracks	Number of tracks.	
ncdiTrackInfos	A List of NERO_TRACK_INFO structures.	

Identifier	Introduced in *NeroAPI* version
NERO_CD_INFO	5.0.3.9
ncdiAvailableEraseModes	5.5.4.7
ncdiUnusedBlocks	5.5.5.8
ncdiMediaType	5.5.9.4
ncdiReserved	5.5.9.4: Reduced size from 30 to 29.
ncdiMediumFlags	6.0.0.10
ncdiLayer0MaxBlocks	6.0.0.19
ncdiTotalCapacity	6.3.0.5
NCDIMF_FINALIZED_DISC	6.6.0.12
ncdiTotalCapacity	6.6.0.11

7.1.13. NERO_CITE_ARGS

This struct can be used to pass additional parameters to NeroCreateIsoTrackEx, in certain cases, e.g.

- if a media shall have two different file systems (e.g. HFS+ CDs), you can provide the second file system with firstRootItem_wrapper.
- if you intend to pass information to be written to the volume descriptor

```
typedef struct tag_NERO_CITE_ARGS {
      int size;
      NERO_ISO_ITEM *firstRootItem;
      NERO_ISO_ITEM *firstRootItem_wrapper;
      const char    *name;
      DWORD          dwBurnOptions;

      const char    *systemIdentifier;
      const char    *volumeSet;
      const char    *publisher;
      const char    *dataPreparer;
      const char    *application;
      const char    *copyright;
      const char    *abstract;
      const char    *bibliographic;
} NeroCITEArgs;
```

Description of structure members	
Size	This parameter will be ignored. Initialize the whole struct with 0. The version of the struct will be taken from the expected *NeroAPI* version.
firstRootItem	If firstRootItem_wrapper is NULL, then firstRootItem is identical to NeroCreateIsoTrackEx's rootItem.
firstRootItem_wrapper	Used to create a wrapper file system. One file system can be embedded in another. Depending on the capabilities of a particular system, the one that can be read will be visible. Unlike the UDF/ISO bridge, both file systems can contain different files, so two pointers to root items are required.
Name	Name of the IsoTrack (volume name).
dwBurnOptions	The same options as used by the NeroCreateIsoTrackEx function.
systemIdentifier	System identifier.
volumeSet	This name is used when multiple media are part of one logical unit.
Publisher	The publisher of this track.
dataPreparer	The preparer of this track.
application	The application that created this track.
Copyright	Copyright file.
Abstract	Abstract file.

Description of structure members	
bibliographic	Bibliographic file.

Identifier	Introduced in *NeroAPI* version
NeroCITEArgs	5.5.9.0
systemIdentifier	5.5.9.26
volumeSet	5.5.9.26
Publisher	5.5.9.26
dataPreparer	5.5.9.26
application	5.5.9.26
Copyright	5.5.9.26
Abstract	5.5.9.26
bibliographic	5.5.9.26
NERO_CITE_ARGS	6.0.0.0: NeroCITEArgs was renamed to NERO_CITE_ARGS.

7.1.14. NERO_CONFIG_RESULT

This is the return type for the NeroAudioGUIConfigureItem function.

```
typedef enum
{
  NCR_CANNOT_CONFIGURE,
  NCR_CHANGED,
  NCR_NOT_CHANGED
} NERO_CONFIG_RESULT;
```

Description of enumerators	
NCR_CANNOT_CONFIGURE	The item cannot be configured.
NCR_CHANGED	The configuration has been changed.
NCR_NOT_CHANGED	The configuration has not been changed.

Identifier	Introduced in *NeroAPI* version
NERO_CONFIG_RESULT	5.5.9.8

7.1.15. NERO_DATA_EXCHANGE

Use PCM, 44.1kHz, Stereo (left channel first), 16 bits per channel, Little Endian Word (LSB first), when exchanging data with the *NeroAPI*.

```
typedef struct tag_NERO_DATA_EXCHANGE
{
  NERO_DATA_EXCHANGE_TYPE ndeType;
  union
  {
    char      ndeFileName[256];
    struct
    {
       DWORD       reserved;
       const char  *ptr;
    } ndeLongFileName;

    NERO_IO   ndeIO;
    NERO_AUDIO_ITEM_INFO     ndeAudioItemInfo;
  } ndeData;
} NERO_DATA_EXCHANGE;
```

Description of structure members	
ndeFileName	Deprecated, use ndeLongFileName.ptr instead.
ndeLongFileName.reserved	Must be 0.
ndeIO	NERO_IO/EOF/ERROR_CALLBACK, data is exchanged with the application directly.
ndeAudioItemInfo	NERO_ET_AUDIO_FILE, data is exchange through audio items, using the plug-in manager.

Identifier	Introduced in *NeroAPI* version
NERO_DATA_EXCHANGE	5.0.3.9
ndeAudioItemInfo	5.5.9.8
ndeLongFileName	6.0.0.0

7.1.16. NERO_DATA_EXCHANGE_TYPE

This enum is used as a member of the NERO_DATA_EXCHANGE struct.

```
typedef enum
{
  NERO_ET_FILE,
  NERO_ET_IO_CALLBACK,
  NERO_ET_MP3,
  NERO_ET_FILE_RAW,
  NERO_ET_AUDIO_FILE
} NERO_DATA_EXCHANGE_TYPE;
```

Description of enumerators	
NERO_ET_FILE	Read/write to/from WAV file.
NERO_ET_IO_CALLBACK	Exchange data with application directly.
NERO_ET_MP3	Read from MP3 file (not for DAE).
NERO_ET_WMA	Read from MS audio file (not for DAE).
NERO_ET_FILE_RAW	For a Freestyle compilation, this and NERO_ET_IO_CALLBACK are the only types allowed at the moment. It will expect files to be in the format as to be written to the disc. This exchange type is valid for freestyle compilations only.
NERO_ET_AUDIO_FILE	Audio file created with the plug-in manager.

Identifier	Introduced in *NeroAPI* version
NERO_DATA_EXCHANGE_TYPE	5.0.3.9
NERO_ET_AUDIO_FILE	5.5.9.8

7.1.17. NERO_DEVICEHANDLE

Is defined as a class pointer for C++ or a void pointer for standard C.

The __cplusplus preprocessor macro determines whether C++ or C is being compiled. This macro is predefined and gives the programmer the opportunity to use more sophisticated C++ constructs where possible, or substitute them with standard C where not.

```
#ifdef  __cplusplus
class CSourceDriveInfo;
typedef CSourceDriveInfo *NERO_DEVICEHANDLE;
#else
typedef void *NERO_DEVICEHANDLE;
#endif
```

Identifier	Introduced in *NeroAPI* version
NERO_DEVICEHANDLE	5.0.3.9

7.1.18. NERO_DISABLE_ABORT_CALLBACK

Tells the main program whether the burn process can be interrupted or not.

```
typedef void (NERO_CALLBACK_ATTR *NERO_DISABLE_ABORT_CALLBACK)(void
*pUserData,BOOL abortEnabled);
```

Identifier	Introduced in *NeroAPI* version
NERO_DISABLE_ABORT_CALLBACK	5.0.3.9

7.1.19. NERO_DEVICEOPTION

Used to get and set special low level options of devices.

```
typedef enum
{
  NERO_DEVICEOPTION_BOOKTYPE_DVDROM = 0,
  NERO_DEVICEOPTION_LAYERSWITCH = 1,
  NERO_DEVICEOPTION_BOOKTYPE_DVDROM_NEXT_WRITE = 2,
  NERO_DEVICEOPTION_BREAK_LAYER = 3,
  NERO_DEVICEOPTION_SPARE_AREA = 4
} NERO_DEVICEOPTION;
```

Description of enumerators	
NERO_DEVICEOPTION_ BOOKTYPE_DVDROM	Change the booktype of a DVD+R and DVD+RW for subsequent writes until next power cycle to DVD-ROM. When used in NeroGetDeviceOption or NeroSetDeviceOption void* is a pointer to BOOL. For setting the booktype to DVD-ROM, set the parameter to TRUE, to reset make it FALSE. In NeroGetDeviceOption, TRUE is returned if changing the booktype to DVD-ROM is enabled for both DVD+R and DVD+RW, FALSE otherwise.
NERO_DEVICEOPTION_LAYER SWITCH	Set the number of blocks after that to switch from layer 0 to layer 1 when writing on a double layer medium. Notes: • The number of blocks must be a multiple of 16. • The layer 0 must be >= totalDataSize/2, because there can never be more data on layer 1 than on layer 0. • In NeroSetDeviceOption value is a pointer to a DWORD variable. • In NeroGetDeviceOption a pointer to a DWORD is returned.

Description of enumerators	
NERO_DEVICEOPTION_ BOOKTYPE_DVDROM_NEXT_ WRITE	Set the booktype of the next DVD+R and DVD+RW that is written to DVD-ROM. This option is useful if you do packet writing. If you call NeroBurn you have to use the NBF_BOOKTYPE_DVDROM flag or NBF_NO_BOOKTYPE_CHANGE flag. void* is a pointer to BOOL in Nero(Set|Get)DeviceOption.
NERO_DEVICEOPTION_ BREAK_LAYER	Set the number of blocks after that to switch from layer 0 to layer 1 when writing on a double layer medium. In difference to NERO_DEVICEOPTION_LAYERSWITCH the layer break is written immediately to the disc, while otherwise it is set during burning. Notes: • The number of blocks must be a multiple of 16 • The layer 0 must be >= totalDataSize/2, because there can never be more data on layer 1 than on layer 0. • In NeroSetDeviceOption value is a pointer to a DWORD variable. • In NeroGetDeviceOption a pointer to a DWORD is returned.
NERO_DEVICEOPTION_ SPARE_AREA	Used for getting or setting the spare area on Blue-ray Discs. In NeroSetDeviceOption <*value> is a pointer to NERO_SPARE_AREA_SETTINGS which will be allocated when next burning. In NeroGetDeviceOption <void*> is a pointer to DWORD specifying the number of allocated spare area sectors.

Identifier	Introduced in *NeroAPI* version
NERO_DEVICEOPTION	5.5.10.7
NERO_DEVICEOPTION_BOOKTYPE_DVDROM_NEXT_ WRITE	6.0.0.24
NERO_DEVICEOPTION_LAYERSWITCH	6.3.1.4
NERO_DEVICEOPTION_BREAK_LAYER	6.3.1.4
NERO_DEVICEOPTION_SPARE_AREA	7.0.5.6

7.1.20. NERO_DLG_WAITCD_MEDIA_INFO

A pointer to this structure will be passed with the DLG_WAITCD_MEDIA_INFO user dialog callback.

```
typedef struct
{
      DWORD ndwmiSize;
      NERO_MEDIA_TYPE   ndwmiLastDetectedMedia;
      NERO_MEDIA_SET    ndwmiRequestedMedia;
      const char *ndwmiLastDetectedMediaName;
      const char *ndwmiRequestedMediaName;
} NERO_DLG_WAITCD_MEDIA_INFO;
```

Identifier	Introduced in *NeroAPI* version
NERO_DLG_WAITCD_MEDIA_INFO	5.5.9.4

7.1.21. NERO_DRIVE_ERROR

Error code describing an error happened during communication with a drive.

This error code is returned by NeroIsDeviceReady. Other functions set an internal error variable to one of these codes if an error occurred. This error can be received with NeroGetLastDriveError.

```
typedef enum
{
   NDE_NO_ERROR  = 0,
   NDE_GENERIC_ERROR = 1,
   NDE_DRIVE_IN_USE = 2,
   NDE_DRIVE_NOT_READY = 3,
   NDE_NO_DRIVE = 4,
   NDE_DISC_NOT_PRESENT = 5,
   NDE_DISC_NOT_PRESENT_TRAY_OPEN = 6,
   NDE_DISC_NOT_PRESENT_TRAY_CLOSED = 7
} NERO_DRIVE_ERROR;
```

Description of structure members	
NDE_NO_ERROR	No error occurred/ drive is ready.
NDE_GENERIC_ERROR	Error, not handled with other enums.
NDE_DRIVE_IN_USE	Drive cannot be locked; maybe another application uses this drive at the moment.
NDE_DRIVE_NOT_READY	Drive is not ready.
NDE_NO_DRIVE	The given device is not available. Probably removed by the user (USB/Firewire).
NDE_DISC_NOT_PRESENT	No medium in drive, status of tray unknown.
NDE_DISC_NOT_PRESENT_ TRAY_	No medium - tray open.
NDE_DISC_NOT_PRESENT_ TRAY_CLOSED	No medium - tray closed.

Identifier	Introduced in *NeroAPI* version
NERO_DRIVE_ERROR	6.0.0.0

7.1.22. NERO_DRIVESTATUS_CALLBACK

This callback informs the application about a drive's status change.

Note: The callback needs to be thread safe, since it might be called from a different thread.

```
typedef void  (NERO_CALLBACK_ATTR *NERO_DRIVESTATUS_CALLBACK)(
                                        int hostID,
                                        int targetID,
                                        NERO_DRIVESTATUS_RESULT result,
                                        void *pUserData);
```

Description	
hostID	Corresponds to nsdiHostAdapterNo.
targetID	Corresponds to nsdiDeviceID of NERO_SCSI_DEVICE_INFO.

Identifier	Introduced in *NeroAPI* version
NERO_DRIVESTATUS_CALLBACK	6.0.0.0

7.1.23. NERO_DRIVESTATUS_TYPE

This enum is used by the NeroRegisterDriveStatusCallback callback.

```
typedef enum
{
   NDT_DISC_CHANGE,
   NDT_IN_USE_CHANGE
} NERO_DRIVESTATUS_TYPE;
```

Description of enumerators	
NDT_DISC_CHANGE	The disc in the drive has been changed. Warning: This change notification is based on Windows notifying about medium changes. If an application has disabled this notification, the callback will not be called. If you want to be sure to recognize all medium changes, you should use timer events and use NeroIsDeviceReady to ask for the drive status.
NDT_IN_USE_CHANGE	The in-use status of the drive has been changed.

Identifier	Introduced in *NeroAPI* version
NERO_DRIVESTATUS_TYPE	6.0.0.0

7.1.24. NERO_DRIVESTATUS_RESULT

This enumeration is used by NERO_DRIVESTATUS_CALLBACK.

```
typedef enum
{
   NDR_DRIVE_IN_USE=0,
   NDR_DRIVE_NOT_IN_USE,
   NDR_DISC_REMOVED,
   NDR_DISC_INSERTED,
   NDR_DRIVE_REMOVED,
   NDR_DRIVE_ADDED
} NERO_DRIVESTATUS_RESULT;
```

Identifier	Introduced in *NeroAPI* version
NERO_DRIVESTATUS_RESULT	6.0.0.0

7.1.25. NERO_FILESYSTEMTRACK_OPTIONS

This structure is passed as a parameter type to the NeroEstimateTrackSize function.

```
typedef struct tag_NERO_FILESYSTEMTRACK_OPTIONS
{
      DWORD                   netsStructureSize;
      void *                  netspCDStamp;
      NERO_MEDIA_TYPE         netsMediaType;
      NERO_DEVICEHANDLE       netsDeviceHandle;
      DWORD                   netsFlags;
#ifdef __cplusplus
      FileSystemContent::
#else //__cplusplus
      struct
#endif//__cplusplus
      IFileSystemDescContainer *netsFSContainer;
      DWORD                   netsFSContainerFlags;
      DWORD                   netsReserved[32];
} NERO_FILESYSTEMTRACK_OPTIONS;
```

Description of structure members	
netsStructureSize	Fill this with sizeof(NERO_FILESYSTEMTRACK_OPTIONS).
netspCDStamp	Point on a CDStamp object when appending to an existing medium, otherwise NULL. This entry needs to be filled out whenever the file system size is to be calculated accurately.
netsMediaType	The media type the file system is to be written to. This entry needs to be filled out whenever the file system size is to be calculated accurately.

Description of structure members	
netsDeviceHandle	Device handle representing the drive the file system is to be written to. This entry needs to be filled out whenever the file system size is to be calculated accurately.
netsFlags	NBF_XXXX that will be used for the recording process.
netsFSContainer	If not NULL, the file system will be created from this object instead of the passed CNeroIsoTrack object. pIsoTrack must be NULL in this case.
netsFSContainerFlags	NCITEF_XXXX flags to be used for file system creation. Used only when netsFSContainer is used to create the file system.
netsReserved	Should be zero.

7.1.26. NERO_FREESTYLE_TRACK

This type is used as a member of NERO_WRITE_FREESTYLE_CD.

```
typedef struct tag_NERO_FREESTYLE_TRACK
{
  DWORD nftStructureSize;
  DWORD nftPauseInBlksBeforeThisTrack;
  DWORD nftNumIndexPositions;
  DWORD nftRelativeIndexBlkPositions[98];
  const char *nftTitle, *nftArtist;
  NERO_DATA_EXCHANGE nftSourceDataExchg;
  DWORD nftLengthInBlocks;
  NERO_TRACKMODE_TYPE nftTracktype;
} NERO_FREESTYLE_TRACK;
```

Description of structure members	
nftStructureSize	Size of this structure, to ensure binary compatibility.
nftPauseInBlksBeforeThisTrack	Pause in blocks before this track.
nftNumIndexPositions	Number of index positions.
nftRelativeIndexBlkPositions[98]	Offsets between one index position and the next one.
nftTitle	Set to NULL if unknown or to be taken from source.
nftArtist	Set to NULL if unknown or to be taken from source.
nftSourceDataExchg	Source for raw track data.
nftLengthInBlocks	Only used for NERO_IO_CALLBACK.
nftTracktype	Specifies track type to be written.

Identifier	Introduced in *NeroAPI* version
NERO_FREESTYLE_TRACK	5.0.3.9

7.1.27. NERO_FS_NAMETYPE

File system name type enumeration.

```
typedef enum
{
      NFSNT_Filename,
      NFSNT_Dirname,
      NFSNT_VolumeLabel
} NERO_FS_NAMETYPE;
```

Description of enumerators	
NFSNT_Filename	File name.
NFSNT_Dirname	Directory name (ISO9660 has different naming conventions for file and directory names.)
NERO_FS_NAMETYPE	Volume label.

Identifier	Introduced in *NeroAPI* version
NERO_FS_NAMETYPE	6.6.0.13

7.1.28. NERO_FS_TYPE

File system type enumeration.

```
typedef enum
{
      NFST_ISO,
      NFST_UDF,
      NFST_JOLIET,
      NFST_JOLIET_RELAXED,
      NFST_UDF_DVDVIDEO,
      NFST_ROCKRIDGE
} NERO_FS_TYPE;
```

Description of enumerators	
NFST_ISO	ISO 9660
NFST_UDF	UDF
NFST_JOLIET	Joliet
NFST_JOLIET_RELAXED	
NFST_UDF_DVDVIDEO	DVD-Video compatible UDF
NFST_ROCKRIDGE	Rockridge

Identifier	Introduced in *NeroAPI* version
NERO_FS_TYPE	6.6.0.13

7.1.29. NERO_IDLE_CALLBACK

During writing, or in several long running functions, control is transferred to the DLL. The application has to provide services and interact with the user via callback functions.

NERO_CALLBACK_ATTR is defined in "NeroUserDialog.h" and ensures that the same conventions are used for passing of parameters. NERO_IDLE_CALLBACK is called regularly during long running activities. Return TRUE if this activity shall be aborted.

```
typedef BOOL (NERO_CALLBACK_ATTR *NERO_IDLE_CALLBACK) (void *pUserData);
```

Identifier	Introduced in *NeroAPI* version
NERO_IDLE_CALLBACK	5.0.3.9

7.1.30. NERO_IMPORT_DATA_TRACK_INFO

This structure is used as a parameter for the NeroImportDataTrack function.

```
typedef struct tag_NERO_IMPORT_DATA_TRACK_INFO
{
  DWORD nidtiSize;
  char *nidtipVolumeName;
} NERO_IMPORT_DATA_TRACK_INFO;
```

Description of structure members	
nidtiSize	Must contain the size of the structure.
nidtipVolumeName	This must be released using NeroFreeMem.

Identifier	Introduced in *NeroAPI* version
NERO_IMPORT_DATA_TRACK_INFO	6.0.0.0

7.1.31. NERO_IMPORT_DATA_TRACK_RESULT

This enum is used as result parameter for the NeroImportDataTrack function.

```
typedef enum
{
      NIDTR_NO_ERROR=0,
      NIDTR_GENERIC_ERROR,
      NIDTR_DRIVE_ERROR,
      NIDTR_READ_ERROR,
      NIDTR_INVALID_FS
} NERO_IMPORT_DATA_TRACK_RESULT;
```

Description of enumerators	
NIDTR_NO_ERROR	No error.
NIDTR_GENERIC_ERROR	Undefined error.
NIDTR_DRIVE_ERROR	Get more details with NeroGetLastDriveError.
NIDTR_READ_ERROR	Error while reading from the disc. Parts of the file system may have been imported nevertheless.
NIDTR_INVALID_FS	Errors in the file system on the disc. Parts of the files system may have been imported nevertheless.

Identifier	Introduced in *NeroAPI* version
NERO_IMPORT_DATA_TRACK_RESULT	6.0.0.0

7.1.32. NERO_IO

NERO_IO is required when the *NeroAPI* exchanges data with the application directly. NERO_IO is used as member of the NERO_DATA_EXCHANGE struct.

```
typedef struct tag_NERO_IO
{
  void  *nioUserData;
  NERO_IO_CALLBACK  nioIOCallback;
  NERO_STATUS_CALLBACK  nioEOFCallback;
  NERO_STATUS_CALLBACK  nioErrorCallback;
} NERO_IO;
```

Description of structure members	
nioUserData	Provide the this-pointer here.
nioIOCallback	See declaration of NERO_IO_CALLBACK.
nioEOFCallback	Shall return TRUE if further IO calls will always fail to transfer any data, i.e. EOF reached.
nioErrorCallback	Shall return TRUE if an error occurred during an IO call.

Identifier	Introduced in *NeroAPI* version
NERO_IO	5.0.3.9

7.1.33. NERO_IO_CALLBACK

Data exchange between an application and the *NeroAPI* is done with a function that gets a pointer to its own structure, a buffer pointer and the amount in bytes to be read or written. It shall return the actual amount of bytes transferred. Other functions indicate that EOF has been reached when reading or a serious error occurred.

```
typedef DWORD (NERO_CALLBACK_ATTR *NERO_IO_CALLBACK)
(void *pUserData, BYTE *pBuffer, DWORD dwLen);
```

Identifier	Introduced in *NeroAPI* version
NERO_IO_CALLBACK	5.0.3.9

7.1.34. NERO_ISO_ITEM

This type is used for ISO track generation. The *NeroAPI* offers functions to create ISO items, copy them, free space used by an item, and creates tracks based on an ISO root item.

```
typedef struct tag_NERO_ISO_ITEM
{
  char fileName[252];
  char *longFileName;
  BOOL isDirectory;
  BOOL isReference;
  char sourceFilePath[252];
  const char *longSourceFilePath;
  char sourceFilePath[256];
  struct tag_NERO_ISO_ITEM *subDirFirstItem;
  struct tag_NERO_ISO_ITEM *nextItem;
  void *userData;
  long dataStartSec;
  __int64 dataLength;
  struct tm entryTime;
  int itemSize;
  struct CImportInfo *importinfo;
} NERO_ISO_ITEM;
```

Description of structure members	
fileName	Deprecated, use longFileName instead.
longFileName	File name on the burnt CD. It will be freed in NeroFreeIsoItem if this item is a reference.
isDirectory	Is this item a directory?
isReference	Is this item a reference to a file/directory of a previous session?

Description of structure members	
sourceFilePath	Deprecated, use longSourceFilePath instead
longSourceFilePath	Path to the file, including file name (ignored for a directory). When recording rockridge, you can set the name of a directory to be used for retrieving rockridge information here.
subDirFirstItem	Point on the first item of the sub directory if the item is a directory. Can be NULL if the directory is empty. (ignored for a file)
nextItem	Next item in the current directory
userData	Can be used to store additional information
dataStartSec	Used to reference a file from a previous session
dataLength	Used to reference a file from a previous session
entryTime	Used to reference a file from a previous session
itemSize	Size of the structure
importinfo	Optional pointer to an object with import information.

Identifier	Introduced in *NeroAPI* version
NERO_ISO_ITEM	5.0.3.9
itemSize	5.5.0.6
Importinfo	5.5.0.6 5.5.7.5: “rockridge” is renamed to “importinfo”
filename	6.0.0.0: Size changed from 256 to 252.
longFileName	6.0.0.0
sourceFilePath	6.0.0.0: Size changed from 256 to 252.
longSourceFilePath	6.0.0.0

7.1.35. NERO_MAJOR_PHASE

This enum is used by NERO_SET_MAJOR_PHASE_CALLBACK. It indicates what major phase the burn process is currently in.

```
typedef enum
{
      NERO_PHASE_UNSPECIFIED                    =-1,
      NERO_PHASE_START_CACHE                    =24,
      NERO_PHASE_DONE_CACHE                     =25,
      NERO_PHASE_FAIL_CACHE                     =26,
      NERO_PHASE_ABORT_CACHE                    =27,
      NERO_PHASE_START_TEST                     =28,
      NERO_PHASE_DONE_TEST                      =29,
      NERO_PHASE_FAIL_TEST                      =30,
      NERO_PHASE_ABORT_TEST                     =31,
      NERO_PHASE_START_SIMULATE                 =32,
      NERO_PHASE_DONE_SIMULATE                  =33,
```

```
      NERO_PHASE_FAIL_SIMULATE                =34,
      NERO_PHASE_ABORT_SIMULATE               =35,
      NERO_PHASE_START_WRITE                  =36,
      NERO_PHASE_DONE_WRITE                   =37,
      NERO_PHASE_FAIL_WRITE                   =38,
      NERO_PHASE_ABORT_WRITE                  =39,
      NERO_PHASE_START_SIMULATE_NOSPD         =61,
      NERO_PHASE_DONE_SIMULATE_NOSPD          =62,
      NERO_PHASE_FAIL_SIMULATE_NOSPD          =63,
      NERO_PHASE_ABORT_SIMULATE_NOSPD         =64,
      NERO_PHASE_START_WRITE_NOSPD            =65,
      NERO_PHASE_DONE_WRITE_NOSPD             =66,
      NERO_PHASE_FAIL_WRITE_NOSPD             =67,
      NERO_PHASE_ABORT_WRITE_NOSPD            =68,
      NERO_PHASE_PREPARE_ITEMS                =73,
      NERO_PHASE_VERIFY_COMPILATION           =78,
      NERO_PHASE_VERIFY_ABORTED               =79,
      NERO_PHASE_VERIFY_END_OK                =80,
      NERO_PHASE_VERIFY_END_FAIL              =81,
      NERO_PHASE_ENCODE_VIDEO                 =82,
      NERO_PHASE_SEAMLESSLINK_ACTIVATED       =87,
      NERO_PHASE_BUP_ACTIVATED                =90,
      NERO_PHASE_CONTINUE_FORMATTING          =99,
      NERO_PHASE_FORMATTING_SUCCESSFUL        =100,
      NERO_PHASE_FORMATTING_FAILED            =101,
      NERO_PHASE_PREPARE_CD                   =105,
      NERO_PHASE_DONE_PREPARE_CD              =106,
      NERO_PHASE_FAIL_PREPARE_CD              =107,
      NERO_PHASE_ABORT_PREPARE_CD             =108,
      NERO_PHASE_DVDVIDEO_DETECTED            =111,
      NERO_PHASE_DVDVIDEO_REALLOC_STARTED     =112,
      NERO_PHASE_DVDVIDEO_REALLOC_COMPLETED   =113
      NERO_PHASE_DVDVIDEO_REALLOC_NOTNEEDED   =114,
      NERO_PHASE_DVDVIDEO_REALLOC_FAILED      =115,
      NERO_PHASE_DRM_CHECK_FAILURE            =169
} NERO_MAJOR_PHASE;
```

Identifier	Introduced in *NeroAPI* version
NERO_MAJOR_PHASE	5.0.3.9
NERO_PHASE_BUP_ACTIVATED	5.5.7.8
NERO_PHASE_DVDVIDEO_DETECTED	5.5.7.8
NERO_PHASE_DVDVIDEO_REALLOC_STARTED	5.5.7.8
NERO_PHASE_DVDVIDEO_REALLOC_COMPLETED	5.5.7.8
NERO_PHASE_CONTINUE_FORMATTING	5.5.8.0
NERO_PHASE_SEAMLESSLINK_ACTIVATED	5.5.8.2
NERO_PHASE_FORMATTING_SUCCESSFUL	5.5.8.2
NERO_PHASE_DVDVIDEO_REALLOC_NOTNEEDED	5.5.9.3
NERO_PHASE_DVDVIDEO_REALLOC_FAILED	5.5.9.3

Identifier	Introduced in *NeroAPI* version
NERO_PHASE_FAIL_CACHE	6.0.0.0
NERO_PHASE_ABORT_CACHE	6.0.0.0
NERO_PHASE_FAIL_TEST	6.0.0.0
NERO_PHASE_ABORT_TEST	6.0.0.0
NERO_PHASE_FAIL_SIMULATE	6.0.0.0
NERO_PHASE_ABORT_SIMULATE	6.0.0.0
NERO_PHASE_FAIL_WRITE	6.0.0.0
NERO_PHASE_ABORT_WRITE	6.0.0.0
NERO_PHASE_FAIL_SIMULATE_NOSPD	6.0.0.0
NERO_PHASE_ABORT_SIMULATE_NOSPD	6.0.0.0
NERO_PHASE_FAIL_WRITE_NOSPD	6.0.0.0
NERO_PHASE_ABORT_WRITE_NOSPD	6.0.0.0
NERO_PHASE_ABORT_WRITE_NOSPD	6.0.0.0
NERO_PHASE_PREPARE_ITEMS	6.0.0.0
NERO_PHASE_VERIFY_COMPILATION	6.0.0.0
NERO_PHASE_VERIFY_ABORTED	6.0.0.0
NERO_PHASE_VERIFY_END_OK	6.0.0.0
NERO_PHASE_VERIFY_END_FAIL	6.0.0.0
NERO_PHASE_FORMATTING_FAILED	6.0.0.0
NERO_PHASE_PREPARE_CD	6.0.0.0
NERO_PHASE_DONE_PREPARE_CD	6.0.0.0
NERO_PHASE_FAIL_PREPARE_CD	6.0.0.0
NERO_PHASE_ABORT_PREPARE_CD	6.0.0.0
NERO_PHASE_DRM_CHECK_FAILURE	6.3.0.6

7.1.36. NERO_SET_MAJOR_PHASE_CALLBACK

This callback tells the application which phase of the burn process *NeroAPI* is currently in.

```
typedef void (NERO_CALLBACK_ATTR *NERO_SET_MAJOR_PHASE_CALLBACK)(void
*pUserData,NERO_MAJOR_PHASE phase,void *reserved);
```

Identifier	Introduced in *NeroAPI* version
NERO_SET_MAJOR_PHASE_CALLBACK	5.0.3.9

7.1.37. NERO_MEDIA_SET

NERO_MEDIA_SET represents a set of several media.

```
typedef DWORD NERO_MEDIA_SET;
```

Identifier	Introduced in *NeroAPI* version
NERO_MEDIA_SET	5.5.8.0

7.1.38. NERO_MEDIA_TYPE

The bit combinations of NERO_MEDIA_TYPE have a relatively uncommon format to ensure binary compatibility.

This might lead to unexpected behavior. For example when checking (mediaType & MEDIA_CDRW) the result will be true, even if mediaType=MEDIA_CDR.

So it is better to test for (mediaType&MEDIA_CDRW) == MEDIA_CDRW.

```
typedef enum tag_NERO_MEDIA_TYPE
{
      MEDIA_NONE                    =     0,
      MEDIA_CD                      =     0x00001,
      MEDIA_DDCD                    =     0x00002,
      MEDIA_DVD_M                   =     0x00004,
      MEDIA_DVD_P                   =     0x00008,
      MEDIA_DVD_RAM                 =     0x00010,
      MEDIA_ML                      =     0x00020,
      MEDIA_MRW                     =     0x00040,

      MEDIA_NO_CDR                  =     0x00080,
      MEDIA_NO_CDRW                 =     0x00100,
      MEDIA_CDRW                    =     MEDIA_CD|MEDIA_NO_CDR,
      MEDIA_CDR                     =     MEDIA_CD|MEDIA_NO_CDRW,
      MEDIA_DVD_ROM                 =     0x00200,
      MEDIA_CDROM                   =     0x00400,

      MEDIA_NO_DVD_M_RW             =     0x00800,
      MEDIA_NO_DVD_M_R              =     0x01000,
      MEDIA_NO_DVD_P_RW             =     0x02000,
      MEDIA_NO_DVD_P_R              =     0x04000,
      MEDIA_DVD_M_R                 =     MEDIA_DVD_M|MEDIA_NO_DVD_M_RW,
      MEDIA_DVD_M_RW                =     MEDIA_DVD_M|MEDIA_NO_DVD_M_R,
      MEDIA_DVD_P_R                 =     MEDIA_DVD_P|MEDIA_NO_DVD_P_RW,
      MEDIA_DVD_P_RW                =     MEDIA_DVD_P|MEDIA_NO_DVD_P_R,
      MEDIA_FPACKET                 =     0x08000,
      MEDIA_VPACKET                 =     0x10000,
      MEDIA_PACKETW                 =     MEDIA_MRW|MEDIA_FPACKET
                                                |MEDIA_VPACKET
      MEDIA_HDB                     =     0x20000
      MEDIA_DVD_P_R9                =     0x40000,
      MEDIA_DVD_M_R9                =     0x80000,
      MEDIA_DVD_ANY_R9              =     MEDIA_DVD_P_R9|MEDIA_DVD_M_R9,
      MEDIA_DVD_ANY                 =
      MEDIA_DVD_M|MEDIA_DVD_P|MEDIA_DVD_RAM|MEDIA_DVD_ANY_R9,
      MEDIA_BD_ROM                  =     0x100000,
      MEDIA_BD_R                    =     0x200000,
      MEDIA_BD_RE                   =     0x400000,
      MEDIA_BD                      =     MEDIA_BD_R | MEDIA_BD_RE,
      MEDIA_BD_ANY                  =     MEDIA_BD|MEDIA_BD_ROM,
      MEDIA_HD_DVD_ROM              =     0x0800000,
      MEDIA_HD_DVD_R                =     0x1000000,
```

```
      MEDIA_HD_DVD_RW                =     0x2000000,
      MEDIA_HD_DVD                   =     MEDIA_HD_DVD_R|MEDIA_HD_DVD_RW,
      MEDIA_HD_DVD_ANY               =     MEDIA_HD_DVD|MEDIA_HD_DVD_ROM,
} NERO_MEDIA_TYPE;
```

Description of enumerators	
MEDIA_NONE	No media present.
MEDIA_CD	CD-R/RW
MEDIA_DDCD	DDCD-R/RW
MEDIA_DVD_M	DVD-R/RW
MEDIA_DVD_P	DVD+RW
MEDIA_DVD_ANY	Any DVD media
MEDIA_DVD_RAM	DVD-RAM
MEDIA_ML	ML (Multi Level disc)
MEDIA_MRW	Mt. Rainier
MEDIA_NO_CDR	Exclude CD-R
MEDIA_NO_CDRW	Exclude CD-RW
MEDIA_CDRW	CD-RW
MEDIA_CDR	CD-R
MEDIA_DVD_ROM	DVD-ROM (non writable)
MEDIA_CDROM	CD-ROM (non writable)
MEDIA_NO_DVD_M_RW	Exclude DVD-RW
MEDIA_NO_DVD_M_R	Exclude DVD-R
MEDIA_NO_DVD_P_RW	Exclude DVD+RW
MEDIA_NO_DVD_P_R	Exclude DVD+R
MEDIA_DVD_M_R	DVD-R
MEDIA_DVD_M_RW	DVD-RW
MEDIA_DVD_P_R	DVD+R
MEDIA_DVD_P_RW	DVD+RW
MEDIA_FPACKET	Fixed Packet writing
MEDIA_VPACKET	Variable Packet writing
MEDIA_PACKETW	A bit mask for packet writing
MEDIA_HDB	HD-Burn
MEDIA_DVD_P_R9	DVD+R Double Layer 9GB
MEDIA_DVD_M_R9	DVD-R Dual Layer 9GB
MEDIA_DVD_ANY_R9	Any DVD Dual/Double Layer 9GB
MEDIA_BD_ROM	Blu-ray Disc ROM
MEDIA_BD_R	Blu-ray Disc Recordable
MEDIA_BD_RE	Any recordable Blu-ray Disc media
MEDIA_BD_ANY	Any Blu-ray Disc media
MEDIA_HD_DVD_ROM	HD DVD ROM
MEDIA_HD_DVD_R	HD DVD Recordable
MEDIA_HD_DVD_RW	HD DVD Rewritable
MEDIA_HD_DVD	Any recordable HD DVD media
MEDIA_HD_DVD_ANY	Any HD DVD media

Identifier	Introduced in *NeroAPI* version
NERO_MEDIA_TYPE	5.5.4.3
MEDIA_NONE	5.5.9.4
MEDIA_NO_CDR	5.5.9.4
MEDIA_NO_CDRW	5.5.9.4
MEDIA_CDRW	5.5.9.4
MEDIA_CDR	5.5.9.4
MEDIA_DVD_ROM	5.5.9.4
MEDIA_CDROM	5.5.9.4
MEDIA_NO_DVD_M_RW	5.5.9.10
MEDIA_NO_DVD_M_R	5.5.9.10
MEDIA_NO_DVD_P_RW	5.5.9.10
MEDIA_NO_DVD_P_R	5.5.9.10
MEDIA_DVD_M_R	5.5.9.10
MEDIA_DVD_M_RW	5.5.9.10
MEDIA_DVD_P_R	5.5.9.10
MEDIA_DVD_P_RW	5.5.9.10
MEDIA_FPACKET	5.5.9.10
MEDIA_VPACKET	5.5.9.10
MEDIA_PACKETW	5.5.9.10
MEDIA_HDB	5.5.10.4
MEDIA_DVD_P_R9	6.0.0.29
MEDIA_DVD_M_R9	6.6.0.8
MEDIA_BD_ROM	6.6.0.8
MEDIA_BD_R	6.6.0.8
MEDIA_BD_RE	6.6.0.8
MEDIA_BD_ANY	6.6.0.8
MEDIA_HD_DVD_ROM	6.6.0.1001
MEDIA_HD_DVD_R	6.6.0.1001
MEDIA_HD_DVD_RW	6.6.0.1001
MEDIA_HD_DVD	6.6.0.1001
MEDIA_HD_DVD_ANY	6.6.0.1001

7.1.39. NERO_MEDIUM_TYPE

This type is obsolete and **should not be used anymore**. Please use NERO_MEDIA_TYPE instead.

```
typedef enum
{
  NMT_UNKNOWN,
  NMT_CD_ROM,
  NMT_CD_RECORDABLE,
  NMT_CD_REWRITEABLE
} NERO_MEDIUM_TYPE;
```

Identifier	Introduced in *NeroAPI* version
NERO_PROGRESS	5.0.3.9
npSetMajorPhaseCallback	5.5.5.8
npSubTaskProgressCallback	5.5.6.6
NERO_MEDIUM_TYPE	5.0.3.9

7.1.40. NERO_BOOK_TYPE

Book types of the the inserted disc. In rare cases, besides DVD-ROM, it may differ from the physical disc type.

```
typedef enum
{
      NBT_UNKNOWN,
      NBT_DVD_ROM,
      NBT_DVD_RAM,
      NBT_DVD_M_R,
      NBT_DVD_P_R,
      NBT_DVD_M_RW,
      NBT_DVD_P_RW,
      NBT_DVD_P_R_DL,
}
NERO_BOOK_TYPE;
```

Description of enumerators	
NBT_UNKNOWN	Unknown
NBT_DVD_ROM	DVD-ROM
NBT_DVD_RAM	DVD-RAM
NBT_DVD_M_R	DVD-R
NBT_DVD_P_R	DVD+R
NBT_DVD_M_RW	DVD-RW
NBT_DVD_P_RW	DVD+RW
NBT_DVD_P_R_DL	DVD+R Dual Layer

Identifier	Introduced in *NeroAPI* version
NERO_BOOK_TYPE	6.6.0.13

7.1.41. NERO_PROGRESS

Is used for passing required callback function pointers to the NeroBurn function. npDisableAbortCallback will be called only if the NBF_DISABLE_ABORT flag is given to the NeroBurn function.

npSubTaskProgressCallback provides the write buffer fill level.

```
typedef struct tag_NERO_PROGRESS
{
  NERO_PROGRESS_CALLBACK  npProgressCallback;
  NERO_ABORTED_CALLBACK  npAbortedCallback;
  NERO_ADD_LOG_LINE_CALLBACK  npAddLogLineCallback;
  NERO_SET_PHASE_CALLBACK  npSetPhaseCallback;
  NERO_DISABLE_ABORT_CALLBACK npDisableAbortCallback;
  NERO_SET_MAJOR_PHASE_CALLBACK npSetMajorPhaseCallback;
  NERO_PROGRESS_CALLBACK npSubTaskProgressCallback;
} NERO_PROGRESS;
```

Identifier	Introduced in *NeroAPI* version
NERO_PROGRESS	5.0.3.9
npSetMajorPhaseCallback	5.5.5.8
npSubTaskProgressCallback	5.5.6.6

7.1.42. NERO_PROGRESS_CALLBACK

This function needs to return TRUE if the user wants to abort.

The application may provide callback functions to set the different parts of this display. All of them may be NULL.

```
typedef BOOL  (NERO_CALLBACK_ATTR *NERO_PROGRESS_CALLBACK)
                        (void*pUserData, DWORD dwProgressInPercent);
```

Identifier	Introduced in *NeroAPI* version
NERO_PROGRESS_CALLBACK	5.0.3.9

7.1.43. NERO_SCSI_DEVICE_INFO

This struct provides information about a device. It is used in NERO_SCSI_DEVICE_INFOS, the return type of NeroGetAvailableDrivesEx. Apart from that, it is a required parameter when opening a device by a call to NeroOpenDevice.

```
typedef struct tag_NERO_SCSI_DEVICE_INFO
{
  char    nsdiDeviceName[32];
  char    nsdiHostAdapterName[4];
  char    *nsdiLongHostAdapterName;
  DWORD   nsdiHostAdapterNo;
  DWORD   nsdiDeviceID;
  NEROAPI_SCSI_DEVTYPE   nsdiDevType;
  char    nsdiDriveLetter;
  DWORD   nsdiCapabilities;
  NERO_SPEED_INFOS  nsdiReadSpeeds;
  NERO_SPEED_INFOS  nsdiWriteSpeeds;
  const void        *nsdiDriver;
  char*   NsdiBufUnderrunProtName[64];
  DWORD   nsdiMandatoryBUPSpeed;
  NERO_MEDIA_SET    nsdiMediaSupport;
  DWORD   nsdiDriveBufferSize;
  DWORD   nsdiDriveError;
  NERO_MEDIA_SET  nsdiMediaReadSupport;
  char    nsdiFirmwareRevision[8];
  DWORD   nsdiReserved[59];
 } NERO_SCSI_DEVICE_INFO;
```

Description of structure members		
nsdiDeviceName	Device name.	
nsdiHostAdapterName	Host Adapter name.	
nsdiLongHostAdapter Name	Unlimited host adapter name. Adapter names can be more than 8 characters long; therefore we define a pointer that has to be freed afterwards with NeroFreeMem. This pointer will be filled only if a specific option is passed to NeroGetAvailableDevicesEx, and the 'old' field will be then zeroed.	
nsdiHostAdapterNo	Host Adapter number.	
nsdiDeviceID	Device ID.	
nsdiDevType	Device type.	
nsdiDriveLetter	Windows drive letter or 0 if not available.	
nsdiCapabilities	drive capabilities:	
	NSDI_ALLOWED	The drive can only be used if this bit is set.
	NSDI_DAO	Can write in DAO.
	NSDI_READ_CD_TEXT	Can read CD text.

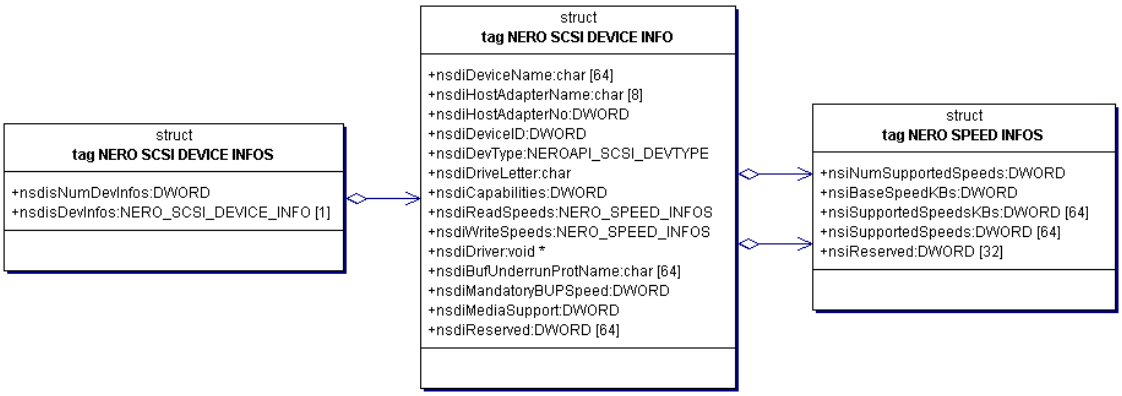
Description of structure members		
	NSDI_VARIABLE_PAUSES_IN_TAO	See natPauseInBlksBeforeThisTrack below.
	NSDI_DAO_WRITE_CD_TEXT	Writes CD text in DAO (see natArtist/Title); never supported in TAO.
	NSDI_BURN_PROOF	Drive can use the burn proof mode. This flag is present for compatibility, better use the NSDI_BUF_UNDERRUN _PROT flag to support other technologies too
	NSDI_IMAGE_RECORDER	Drive is the image recorder.
	NSDI_UNDETECTED	
	NSDI_IDE_BUS	
	NSDI_SCSI_BUS	
	NSDI_BUF_UNDERRUN_PROT	Drive has a buffer underrun protection feature (not necessary Burn Proof)
	NSDI_RESERVED	Must not be used.
	NSDI_RESERVED2	Must not be used.
	NSDI_ALLOW_CHANGE_ BOOKTYPE	DVD recorder can change booktype of burned medium.
	NSDI_DVDPLUSVR_SUPPORTED	This recorder can write DVD+VR.
	NSDI_OVERBURN_SUPPORTED	The device supports overburn.
	NSDI_RESERVED3	Must not be used.
	NSDI_BURN_30MM_AT_LEAST	Device can burn at least 30 mm to be compatible with DVD-ROM.
	NSDI_RESERVED4	Must not be used.
	NSDI_ULTRARAW_96	Device UltraRaw 96 support.
	NSDI_RESERVED5	Must not be used.
nsdiReadSpeeds	See declaration of NERO_SPEED_INFOS.	
nsdiWriteSpeeds	See declaration of NERO_SPEED_INFOS.	
nsdiDriver	Opaque identifier of the internal driver, required by NeroOpenDevice.	
nsdiBufUnderrunProt Name	Buffer underrun protection technology name The string will be empty if the technology has no name	
nsdiMandatoryBUPSp eed	It is highly recommended to enable buffer underrun protection when burning at this speed or faster. Contains 0 if there is no recommendation.	
nsdiMediaSupport	Bit field of supported media (constructed with the NERO_MEDIA_TYPE enum).	
nsdiDriveBufferSize	Drive buffer size (internal) in KB.	

Description of structure members	
nsdiDriveError	Contains a NERO_DRIVE_ERROR that occurred during generating the information. If it differs from NDE_NO_ERROR, some information like the drive capabilities or the speeds might be wrong. NeroIsDeviceReady can be used to check if the drive is ready later and update the device information with NeroUpdateDeviceInfo. NDE_DISC_NOT_PRESENT* errors can be ignored.
nsdiMediaRead Support	Bit field of supported readable media (constructed with the NERO_MEDIA_TYPE enum).
nsdiFirmwareRevision	Firmware Revision of the device. Not less than 4 chars, but with a \0 trailing char.
nsdiReserved	Should be zero.

Identifier	Introduced in *NeroAPI* version
NERO_SCSI_DEVICE_INFO	5.0.3.9
NSDI_BUF_UNDERRUN_PROT	5.5.0.6
nsdiBufUnderrunProtName	5.5.0.6
nsdiMandatoryBUPSpeed	5.5.3.2
nsdiMediaSupport	5.5.4.1 5.5.8.0: Changed type from DWORD to NERO_MEDIA_SET
nsdiDriver	5.5.9.4: Changed from void* to const void*
nsdiDriveBufferSize	5.5.9.4
NSDI_RESERVED2	5.5.10.7
NSDI_ALLOW_CHANGE_BOOKTYPE	5.5.10.7
NSDI_DVDPLUSVR_SUPPORTED	6.0.0.0
nsdiDriveError	6.0.0.0
nsdiMediaReadSupport	6.0.0.8
nsdiLongHostAdapterName	6.6.0.10
NSDI_OVERBURN_SUPPORTED	6.6.0.10
nsdiFirmwareRevision	6.6.0.10
NSDI_BURN_30MM_AT_LEAST	6.6.1.0
NSDI_ULTRARAW_96	6.6.1.0

7.1.44. NERO_SCSI_DEVICE_INFOS

Used to create a list of NERO_SCSI_DEVICE_INFO structures. It is the return type of NeroGetAvailableDrivesEx.




```
typedef struct tag_NERO_SCSI_DEVICE_INFOS
{
  DWORD   nsdisNumDevInfos;
  NERO_SCSI_DEVICE_INFO   nsdisDevInfos[1];
} NERO_SCSI_DEVICE_INFOS;
```

Description of structure members	
nsdisNumDevInfos	Number of entries in nsdisDevInfos.
nsdisDevInfos	See declaration of NERO_SCSI_DEVICE_INFO.

Identifier	Introduced in *NeroAPI* version
NERO_SCSI_DEVICE_INFOS	5.0.3.9

7.1.45. NERO_SET_PHASE_CALLBACK

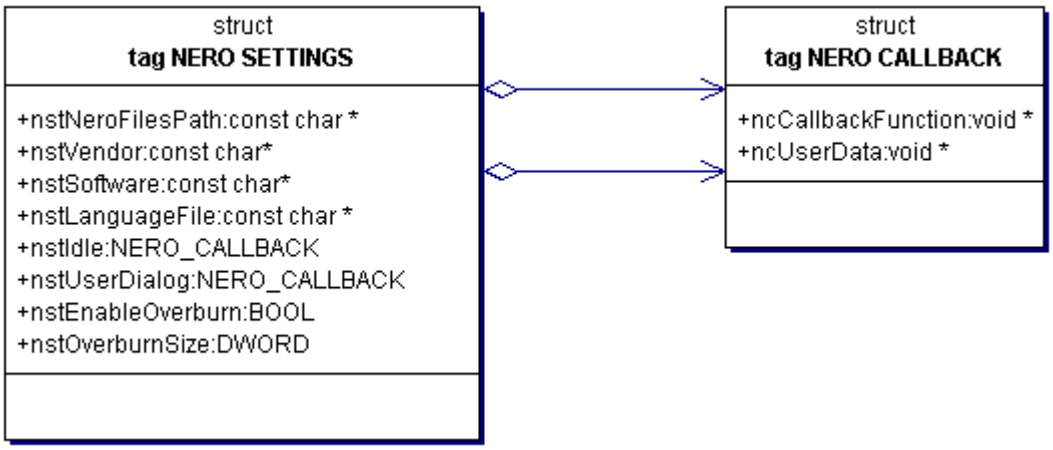
Set the phase line. The text pointer becomes invalid after returning from this function.

```
typedef void  (NERO_CALLBACK_ATTR *NERO_SET_PHASE_CALLBACK)(void
*pUserData, const char *text);
```

Identifier	Introduced in *NeroAPI* version
NERO_SET_PHASE_CALLBACK	5.0.3.9

7.1.46. NERO_SETTINGS

This struct needs to be passed when calling the NeroInit function.




```
typedef struct tag_NERO_SETTINGS
{
  const char  *nstNeroFilesPath;
  const char  *nstVendor, *nstSoftware;
  const char   *nstLanguageFile;
  NERO_CALLBACK   nstIdle;
  NERO_CALLBACK   nstUserDialog;
  BOOL    nstEnableOverburn;
  DWORD   nstOverburnSize;
} NERO_SETTINGS;
```

Description of structure members	
nstNeroFilesPath	Directory name with trailing '\' of where to find the additional Nero DLL and text files.
nstVendor	Path for registry setting. Use “ahead”.
nstSoftware	Path for registry settings. Use "Nero - Burning Rom" for Nero application's settings.

Description of structure members	
nstLanguageFile	Name of the Nero language file; relative to nstNeroFilesPath (e.g. "Nero.txt")
nstIdle	NERO_IDLE_CALLBACK, may be NULL
nstUserDialog	NERO_USER_DIALOG, must not be NULL, see "NeroUserDialog.h" for details
nstEnableOverburn	Overburn settings: Overburning (writing more than the nominal capacity of a disc) is allowed if all of this is true: NstEnableOverburn == TRUE NstOverburnSize >= amount of required blocks for compilation The drive supports it DAO is used. Even then, overburning has to be acknowledged via callback (see DLG_OVERBURN in "NeroUserDialog.h").
nstOverburnSize	In blocks

Identifier	Introduced in *NeroAPI* version
NERO_SETTINGS	5.0.3.9

7.1.47. NERO_SPARE_AREA_SETTINGS

Spare area allocation for BD-R/RE.

```
typedef enum
{
      SAS_NO_ACTION     = 0,
      SAS_ALLOCATE      = 1,
      SAS_NO_ALLOCATE   = 2
}
NERO_SPARE_AREA_SETTINGS;
```

Description of enumerators	
SAS_NO_ACTION	Do nothing by default.
SAS_ALLOCATE	Allocate default spare area.
SAS_NO_ALLOCATE	Do not allocate spare area.

Identifier	Introduced in *NeroAPI* version
NERO_SPARE_AREA_SETTINGS	7.0.5.6

7.1.48. NERO_SPEED_INFOS

This struct will be returned by NeroGetAvailableSpeeds. Two instances of it are used in the NERO_SCSI_DEVICE_INFO struct, for read and write speeds that a particular device supports.

```
typedef struct tag_NERO_SPEED_INFOS
{
  DWORD  nsiNumSupportedSpeeds;
  DWORD nsiBaseSpeedKBs;
  DWORD nsiSupportedSpeedsKBs[64];
  DWORD nsiSupportedSpeeds[64];
  DWORD nsiReserved[32];
} NERO_SPEED_INFOS;
```

Description of structure members	
nsiNumSupportedSpeeds	1 if the speed cannot be changed.
nsiBaseSpeedKBs	Speed corresponding to 1X for the selected media in KB/s.
nsiSupportedSpeedsKBs	List of possible speeds in KB/s
nsiSupportedSpeeds	List of possible speeds in multiple of 150KB/s (1X for CD) (present for compatibility)
nsiReserved	Reserved for future use.

Identifier	Introduced in *NeroAPI* version
NERO_SPEED_INFOS	5.0.3.9

7.1.49. NERO_STATUS_CALLBACK

This callback is used as a part of the data exchange between the *NeroAPI* and an application.

```
typedef BOOL (NERO_CALLBACK_ATTR *NERO_STATUS_CALLBACK)
(void *pUserData);
```

Identifier	Introduced in *NeroAPI* version
NERO_STATUS_CALLBACK	5.0.3.9

7.1.50. NERO_TEXT_TYPE

This type is used by the NERO_ADD_LOG_LINE_CALLBACK to indicate the nature the textual information.

```
typedef enum
{
  NERO_TEXT_INFO,
  NERO_TEXT_STOP,
  NERO_TEXT_EXCLAMATION,
  NERO_TEXT_QUESTION,
```

```
  NERO_TEXT_DRIVE,
  NERO_TEXT_FILE,
  NERO_TEXT_UNSPECIFIED
} NERO_TEXT_TYPE;
```

Description of enumerators	
NERO_TEXT_INFO	Informative text.
NERO_TEXT_STOP	Some operation stopped prematurely
NERO_TEXT_EXCLAMATION	Important information.
NERO_TEXT_QUESTION	A question which requires an answer.
NERO_TEXT_DRIVE	A message concerning a CD-ROM drive or recorder.
NERO_TEXT_FILE	A message concerning a file.
NERO_TEXT_UNSPECIFIED	No type specified.

Identifier	Introduced in *NeroAPI* version
NERO_TEXT_TYPE	5.0.3.9

7.1.51. NERO_TRACK_INFO

A list of this type is contained in NERO_CD_INFO to provide details about every track.

```
typedef struct tag_NERO_TRACK_INFO
{
  DWORD  ntiSessionNumber;
  DWORD   ntiTrackNumber;
  NERO_TRACK_TYPE  ntiTrackType;
  DWORD  ntiTrackStartBlk;
  DWORD  ntiTrackLengthInBlks;
  char  ntiArtist[65];
  char  ntiTitle[65];
  char  ntiISRC[13];
  DWORD  ntiBlockSize;
  DWORD  ntiFSType;
  DWORD  ntiReserved[27];
} NERO_TRACK_INFO;
```

Description of structure members	
ntiSessionNumber	Session Number.
ntiTrackNumber	Track Number.
ntiTrackType	Track Type (Audio, Data, Unknown).
ntiTrackStartBlk	Start Block of Track.
ntiTrackLengthInBlks	Length of Track in Blocks.
ntiArtist[65]	Name of Artist for Audio Tracks.
ntiTitle[65]	Title of Song for Audio Tracks.
ntiISRC[13]	If NGCDI_READ_ISRC is present: 12 char ISRC (International Standard Recording Code) + terminator.
ntiBlockSize	Size of one block in bytes.
ntiFSType	Type of file systems used in tracks. This can be a combination of multiple file systems.
ntiReserved	Should be zero.

Identifier	Introduced in *NeroAPI* version
NERO_TRACK_INFO	5.5.8.3
ntiISRC	5.5.8.4
ntiBlockSize	6.0.0.0
ntiFSType	6.6.0.3

7.1.52. NERO_TRACK_TYPE

This enum is a member of the NERO_TRACK_INFO struct.

```
typedef enum
{
  NTT_UNKNOWN,
  NTT_DATA,
  NTT_AUDIO
} NERO_TRACK_TYPE;
```

Identifier	Introduced in *NeroAPI* version
NERO_TRACK_TYPE	5.0.3.9

7.1.53. NERO_TRACK_FS_TYPE

Determines the type of file system present in a particular track. This enumeration is used by the ntiFSType member of NERO_TRACK_INFO.

```
typedef enum
{
      NTFST_UNKNOWN     = 0,
      NTFST_ISO         = (1<<0),
      NTFST_UDF         = (1<<1),
      NTFST_HFS         = (1<<2),
      NTFST_FAT         = (1<<3)
      NTFST_DATA        = (1<<4),
      NTFST_AUDIO       = (1<<5),
      NTFST_CDRFS       = (1<<6),
      NTFST_DIRECTCD    = (1<<7),
      NTFST_PACKETCD    = (1<<8),
      NTFST_JOLIET      = (1<<9),
      NTFST_BOOTABLE    = (1<<10),
      NTFST_DATA_MODE_1 = (1<<11),
      NTFST_DATA_MODE_2 = (1<<12),
}
NERO_TRACK_FS_TYPE;
```

Identifier	Introduced in *NeroAPI* version
NERO_TRACK_FS_TYPE	6.6.0.3
NTFST_DATA	6.6.1.0
NTFST_AUDIO	6.6.1.0
NTFST_CDRFS	6.6.1.0
NTFST_DIRECTCD	6.6.1.0
NTFST_PACKETCD	6.6.1.0
NTFST_JOLIET	6.6.1.0
NTFST_BOOTABLE	6.6.1.0

Identifier	Introduced in *NeroAPI* version
NTFST_DATA_MODE_1	6.6.1.0
NTFST_DATA_MODE_2	6.6.1.0

7.1.54. NERO_TRACKMODE_TYPE

This type is contained in NERO_FREESTYLE_TRACK to specify the track type that will be written.

```
typedef enum
{
  NERO_TRACKMODE_MODE1,
  NERO_TRACKMODE_MODE2_FORM1,
  NERO_TRACKMODE_AUDIO
} NERO_TRACKMODE_TYPE;
```

Description of enumerators	
NERO_TRACKMODE_MODE1	2048 Bytes per sector data track
NERO_TRACKMODE_MODE2_FORM1	2048 Bytes per sector, used for multisession
NERO_TRACKMODE_AUDIO	2352 Bytes per sector, standard audio track

Identifier	Introduced in *NeroAPI* version
NERO_TRACKMODE_TYPE	5.0.3.9

7.1.55. NERO_VIDEO_ITEM_TYPE

This enum is used in NERO_VIDEO_ITEM to determine the format of the video data.

```
typedef enum
{
  NERO_MPEG_ITEM,
  NERO_JPEG_ITEM,
  NERO_NONENCODED_VIDEO_ITEM,
} NERO_VIDEO_ITEM_TYPE;
```

Description of enumerators	
NERO_MPEG_ITEM	Item is of MPEG type.
NERO_JPEG_ITEM	Item is of JPEG type.
NERO_NONENCODED_VIDEO_ITEM	The source file name will be an AVI file which will be encoded into MPG by the *NeroAPI*.

Identifier	Introduced in *NeroAPI* version
NERO_VIDEO_ITEM_TYPE	5.0.3.9
NERO_NONENCODED_VIDEO_ITEM	5.5.7.8
NERO_DIB_ITEM	6.0.0.24: Removed

7.1.56. NERO_VIDEO_ITEM

A list of NERO_VIDEO_ITEM structs is contained in NERO_WRITE_VIDEO_CD.

```
typedef struct tag_NERO_VIDEO_ITEM
{
  DWORD nviPauseAfterItem;
  char nviSourceFileName[250];
  const char *nviLongSourceFileName;
  NERO_VIDEO_ITEM_TYPE nviItemType;
} NERO_VIDEO_ITEM;
```

Description of structure members	
nviPauseAfterItem	Pause in number of blocks (75 blocks = 1 second).
nviSourceFileName	Deprecated, use nviLongSourceFileName instead.
nviLongSourceFileName	MPG, JPG or AVI file.
nviItemType	Callback functions can only be used for MPG files.

Identifier	Introduced in *NeroAPI* version
NERO_VIDEO_ITEM	5.0.3.9
nviData	6.0.0.0 Removed.
nviSourceFileName	6.0.0.0: Size changed from 236 to 250.

7.1.57. NERO_VIDEO_RESOLUTION

Used by the NERO_WRITE_VIDEO_CD structure.

```
typedef enum
{
      NERO_VIDEO_RESOLUTION_PAL = 0,
      NERO_VIDEO_RESOLUTION_NTSC = 1
} NERO_VIDEO_RESOLUTION;
```

Identifier	Introduced in *NeroAPI* version
NERO_VIDEO_RESOLUTION	6.0.0.17

7.1.58. NERO_VMS_INFO

Virtual multisession is a technique to allow writing multisession discs on media types that do not support normal multisession, e.g. DVD-/+RW.

This structure is the return type of NeroGetVMSInfo.

```
typedef struct tag_NERO_VMS_INFO
{
      DWORD             nvmsiNextWritableAddress;
      DWORD             nvmsiNumSessions;
      DWORD             nvmsiReserved[32];
      NERO_VMSSESSION   nvmsiSessionInfo[1];
} NERO_VMS_INFO;
```

Description of structure members	
nvmsiNextWritableAddress	The next writable address of the media, it may be used for free space calculation.
nvmsiNumSessions	Number of sessions stored on the VMS medium.
nvmsiReserved	Should be zero.
nvmsiSessionInfo	One entry per session.

Identifier	Introduced in *NeroAPI* version
NERO_VMS_INFO	6.0.0.10

7.1.59. NERO_VMSSESSION

This structure describes a single session entry and is used as member in NERO_VMS_INFO.

```
typedef struct tag_NERO_VMSSESSION
{
      char        nvmssSessionName[256];
      struct tm   nvmssCreationTime;
      DWORD       nvmssNextWritableAddress;
      DWORD       nvmssReserved[32];
} NERO_VMSSESSION;
```

Description of structure members	
nvmssSessionName	The name of the session (volume name).
nvmssCreationTime	The creation time of the session.
nvmssNextWritableAddress	The first block that is not occupied by this session.
nvmssReserved	Should be zero.

Identifier	Introduced in *NeroAPI* version
NERO_VMSSESSION	6.0.0.10

7.1.60. NERO_WAITCD_TYPE

This enum is used by the NERO_USER_DIALOG callback and the NeroGetLocalizedWaitCDTexts function.

Due to historical reasons, the enum type and some of its enumerators refer to CD media. Please bear in mind that this type can be used with DVD media as well!

```
typedef enum
{
      NERO_WAITCD_WRITE,
      NERO_WAITCD_SIMULATION,
      NERO_WAITCD_AUTOEJECTLOAD,
      NERO_WAITCD_REINSERT,
      NERO_WAITCD_NEXTCD,
      NERO_WAITCD_ORIGINAL,
      NERO_WAITCD_WRITEPROTECTED,
      NERO_WAITCD_NOTENOUGHSPACE,
      NERO_WAITCD_NEWORIGINAL,
      NERO_WAITCD_EMPTYCD,
      NERO_WAITCD_WRITE_EMPTY,
      NERO_WAITCD_SIMULATION_EMPTY,
      NERO_WAITCD_WRITEWAVE,
      NERO_WAITCD_MULTISESSION,
      NERO_WAITCD_MULTISESSION_SIM,
      NERO_WAITCD_MULTI_REINSERT,
      NERO_WAITCD_DISCINFOS_FAILED,
      NERO_WAITCD_MEDIUM_UNSUPPORTED,
      NERO_WAITCD_AUTOEJECTLOAD_VER,
      NERO_WAITCD_REINSERT_VER,
      NERO_WAITCD_NOFORMAT,
      NERO_WAITCD_WRONG_MEDIUM,
      NERO_WAITCD_WAITING,
      NERO_WAITCD_EMPTYCDRW,
      NERO_WAITCD_NOTENOUGHSPACERW,
      NERO_WAITCD_NOTENOUGHSPACE_80MIN,
      NERO_WAITCD_ISOMULTISESSION_DVDMR9_WARNING,
      NERO_WAITCD_UDFMULTISESSION_DVDMR9_LJR,
      NERO_WAITCD_MAX
} NERO_WAITCD_TYPE;
```

Description of enumerators	
NERO_WAITCD_WRITE	"Please insert the disc to write to..."
NERO_WAITCD_SIMULATION	"Please insert a disc to use during simulation... (Nothing will be written on the disc.)”
NERO_WAITCD_ AUTOEJECTLOAD	"Please do not remove the disc! Your recorder requires this eject between simulation and burning. The disc will be reloaded automatically before continuing with burning...”

Description of enumerators	
NERO_WAITCD_REINSERT	"Please do not remove the disc! Your recorder requires this eject between simulation and burning. Please reinsert the disc..."
NERO_WAITCD_NEXTCD	"Please remove the disc and insert the next recordable disc to write to...”
NERO_WAITCD_ORIGINAL	"Please insert the original disc."
NERO_WAITCD_ WRITEPROTECTED	"This disc is not writable. Please insert a writable disc..."
NERO_WAITCD_ NOTENOUGHSPACE	"There is not enough space to burn this compilation onto this disc. Please insert another disc that provides more space..."
NERO_WAITCD_ NEWORIGINAL	"The disc is blank, invalid nor a multisession disc. Please insert original disc.”
NERO_WAITCD_EMPTYCD	"The disc is not empty. Please insert an empty disc."
NERO_WAITCD_WRITE_ EMPTY	"Please insert an empty disc to write to..."
NERO_WAITCD_SIMULATION_ EMPTY	“Please insert an empty disc to use during simulation... (Nothing will be written on the disc).”
NERO_WAITCD_WRITEWAVE	"The disc is blank. Please insert original disc…”
NERO_WAITCD_ MULTISESSION	"Nero is checking for the disc, please wait ... To burn this multisession compilation you need the disc, which contains the previous backup sessions. Please insert this disc if you haven't done it before.”
NERO_WAITCD_ MULTISESSION_SIM	"To simulate this multisession compilation you need the disc, which contains the previous backup sessions. Please insert this disc. (Nothing will be written on disc)."
NERO_WAITCD_MULTI_ REINSERT	"Please do not remove the disc! Your recorder requires this eject between simulation and burning. Please reinsert the same Multisession disc..."
NERO_WAITCD_DISCINFOS_ FAILED	"Disc analysis failed. The error log contains more information about the reason."
NERO_WAITCD_MEDIUM_ UNSUPPORTED	"The recorder does not support this type of media! Please insert a correct disc to write to..."
NERO_WAITCD_ AUTOEJECTLOAD_VER	"Please do not remove the disc! Your recorder requires that the disc be ejected between burning and verification. The disc will be reloaded automatically when burning is to continue..."
NERO_WAITCD_REINSERT_ VER	"Please do not remove the disc! Your recorder requires that the disc be ejected between burning and verification. Please reinsert the disc...”
NERO_WAITCD_NOFORMAT	"The disc is not formatted. Please insert a formatted disc."
NERO_WAITCD_WRONG_ MEDIUM	"Sorry, your compilation cannot be written on this kind of disc. Please insert a disc of the correct type or modify the settings of your compilation to make them compatible with the current disc."
NERO_WAITCD_WAITING	"--- Accessing disc, please wait ---"
NERO_WAITCD_EMPTYCDRW	"The disc is not empty."
NERO_WAITCD_	"There is not enough space to burn the compilation onto this

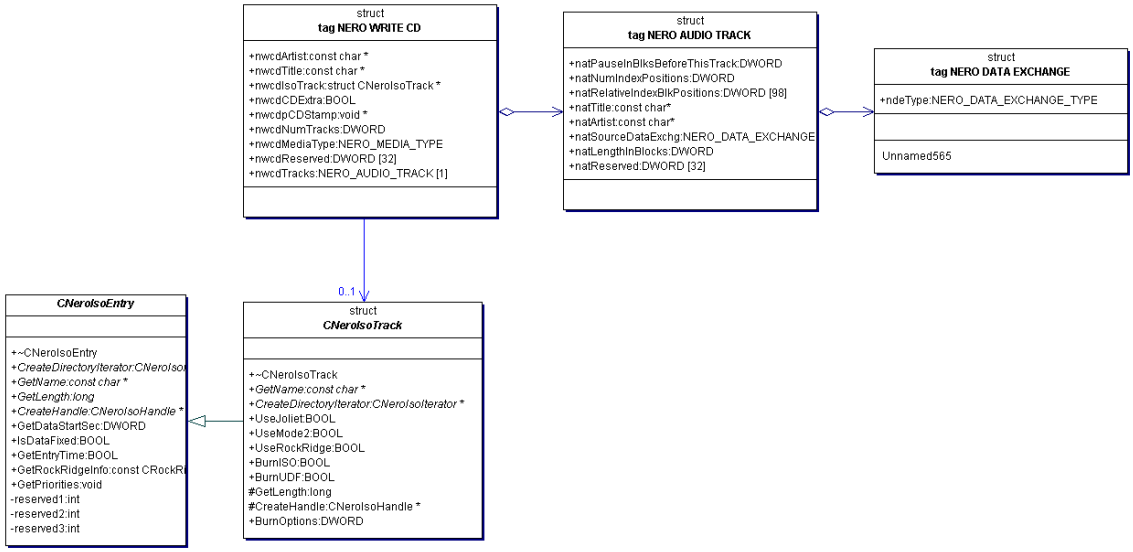
Description of enumerators	
NOTENOUGHSPACERW	disc."
NERO_WAITCD_ NOTENOUGHSPACE_80MIN	"There is not enough space to burn the compilation onto this disc. Please insert a 80min/700MB media"
NERO_WAITCD_MAX	"unknown NERO_WAITCD_TYPE"
NERO_WAITCD_ ISOMULTISESSION_DVDMR9_ WARNING	"Can write multisession or unfinalized DVD-R DL in UDF format only."
NERO_WAITCD_ UDFMULTISESSION_DVDMR9_ LJR	"Can write multisession or unfinalized DVD-R DL on recorders with Layer Jump Recording support only."

Identifier	Introduced in *NeroAPI* version
NERO_WAITCD_WRONG_MEDIUM	5.5.5.6
NERO_WAITCD_WAITING	5.5.10.26
NERO_WAITCD_EMPTYCDRW	6.0.0.20
NERO_WAITCD_NOTENOUGHSPACERW	6.0.0.20
NERO_WAITCD_NOTENOUGHSPACE_80MIN	6.0.0.20
NERO_WAITCD_ISOMULTISESSION_DVDMR9_ WARNING	6.6.0.9
NERO_WAITCD_UDFMULTISESSION_DVDMR9_LJR	6.6.0.9

7.1.61. NERO_WRITE_CD

NERO_WRITE_CD is passed to the NeroBurn function in the pWriteCD parameter, when burning ISO/Audio media.

Due to historical reasons, this type refers to CD media. Please bear in mind that it can be used with DVD media as well!




```
typedef struct tag_NERO_WRITE_CD
{
  const char *nwcdArtist;
  const char *nwcdTitle;
  struct CNeroIsoTrack *nwcdIsoTrack;
  BOOL nwcdCDExtra;
  void *nwcdpCDStamp;
  DWORD nwcdNumTracks;
  NERO_MEDIA_TYPE nwcdMediaType;
  BOOL nwcdAudioMaster;
  DWORD nwcdReserved[31];
  NERO_AUDIO_TRACK nwcdTracks[1];
} NERO_WRITE_CD;
```

Description of structure members	
nwcdArtist	May be NULL.
nwcdTitle	May be NULL.
nwcdIsoTrack	If not NULL, then the disc will have an ISO track - please refer to "NeroIsoTrack.h".
nwCDExtra	If TRUE and nwcdIsoTrack not NULL, then the resulting CD will have audio in the first session and the data track in the second, however, currently the *NeroAPI* does not add any of

Description of structure members	
	the special CD Extra files to the data track. There should be at least one audio track, otherwise this flag will be ignored.
nwcdpCDStamp	Point on a CDStamp object if a particular CD is requested, otherwise NULL.
nwcdNumTracks	Number of Tracks.
nwcdMediaType	Media on which the data should be written.
nwcdAudioMaster	Create an Audio Master CD (if the recorder supports it).
nwcdReserved[31]	
nwcdTracks	See declaration of NERO_AUDIO_TRACK.

Identifier	Introduced in *NeroAPI* version
NERO_WRITE_CD	5.0.3.9
nwcdMediaType	5.5.4.3

7.1.62. NERO_WRITE_FILE_SYSTEM_CONTENT

This type is used when burning an IFileSystemDescContainer.

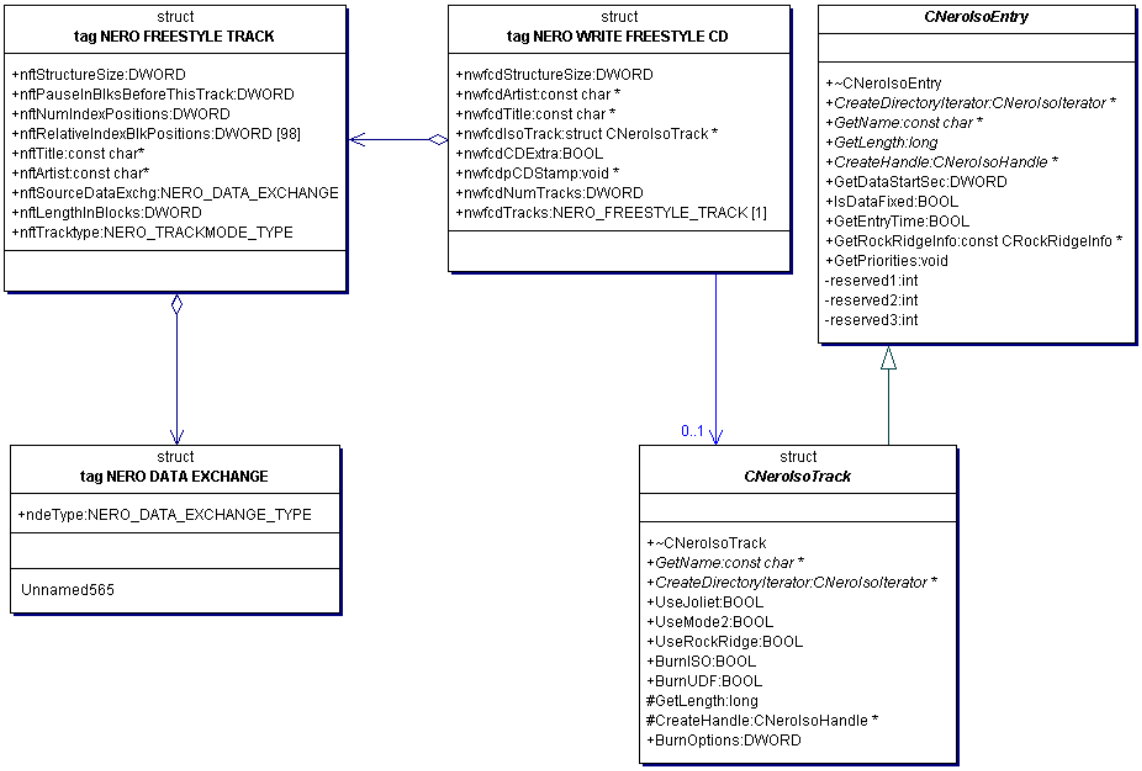
```
typedef struct tag_NERO_WRITE_FILE_SYSTEM_CONTAINER
{
      DWORD nwfscSize;
#ifdef __cplusplus
FileSystemContent::
#else
struct
#endif
      IFileSystemDescContainer *nwfscFSContainer;
      NERO_MEDIA_TYPE   nwfscMediaType;
      DWORD nwfscBurnOptions;
      DWORD nwfscReserved[32];
} NERO_WRITE_FILE_SYSTEM_CONTENT;
```

Description of structure members	
nwfscSize	fill this with sizeof(NERO_WRITE_FILE_SYSTEM_CONTENT)
nwfscFSContainer	A pointer to the IFileSystemDescContainer object.
nwfscMediaType	Media on which the data should be written
nwfscBurnOptions	Combination of NCITEF flags
nwfscReserved	Should be zero

Identifier	Introduced in *NeroAPI* version
NERO_WRITE_FILE_SYSTEM_CONTENT	5.5.6.0

7.1.63. NERO_WRITE_FREESTYLE_CD

This structure will allow you to write any type of CD Layout, e.g. containing a raw data track at the beginning of the disc instead of a self-made ISO/UDF file system. This is good for writing .iso images as can be downloaded everywhere on the net.




```
typedef struct
{
  DWORD nwfcdStructureSize;
  const char *nwfcdArtist;
  const char *nwfcdTitle;
  struct CNeroIsoTrack *nwfcdIsoTrack;
  BOOL nwfcdCDExtra;
  void *nwfcdpCDStamp;
  DWORD nwfcdNumTracks;
  DWORD nwfcdBurnOptions;

#ifdef __cplusplus
FileSystemContent::
#else //__cplusplus
struct
#endif//__cplusplus

  IFileSystemDescContainer *nwfcdFSContainer
  NERO_MEDIA_TYPE nwfcdMediaType;
  DWORD nwfcdReserved[32];
```

```
  NERO_FREESTYLE_TRACK nwfcdTracks[1];
} NERO_WRITE_FREESTYLE_CD;
```

Description of structure members	
nwfcdStructureSize	Fill this with sizeof(NERO_WRITE_FREESTYLE_CD).
nwfcdArtist	may be NULL.
nwfcdTitle	may be NULL.
nwfcdIsoTrack	If not NULL, then the disc will have an ISO track - please refer to the “ISO Track Classes” description.
nwfcdCDExtra	If TRUE and nwfcdIsoTrack not NULL, then the resulting CD will have audio in the first session and the data track in the second, however, currently the *NeroAPI* does not add any of the special CD Extra files to the data track.
nwfcdpCDStamp	Point to a CDStamp object if a particular CD is requested, otherwise NULL.
nwfcdNumTracks	Number of tracks.
nwfcdBurnOptions	Combination of NCITEF flags. Ignored if nwfcdFSContainer is NULL.
nwfcdFSContainer	If not NULL, then the disc will have an ISO track described by this container. nwfcdIsoTrack must be NULL; otherwise the container will be ignored.
nwfcdMediaType	Media on which the data should be written.
nwfcdTracks[1]	List of NERO_FREESTYLE_TRACKs.
nwfcdReserved[32]	Should be zero.

Identifier	**Introduced in *NeroAPI* version**
NERO_WRITE_FREESTYLE_CD	5.0.3.9
nwfcdBurnOptions	5.5.9.1
nwfcdFSContainer	5.5.9.1
nwfcdMediaType	5.5.9.1
nwfcdReserved	5.5.9.1

7.1.64. NERO_WRITE_IMAGE

Used when burning an image.

nwiImageLongFileName contains the name of the image file to burn. Supported formats are NRG, ISO and CUE.

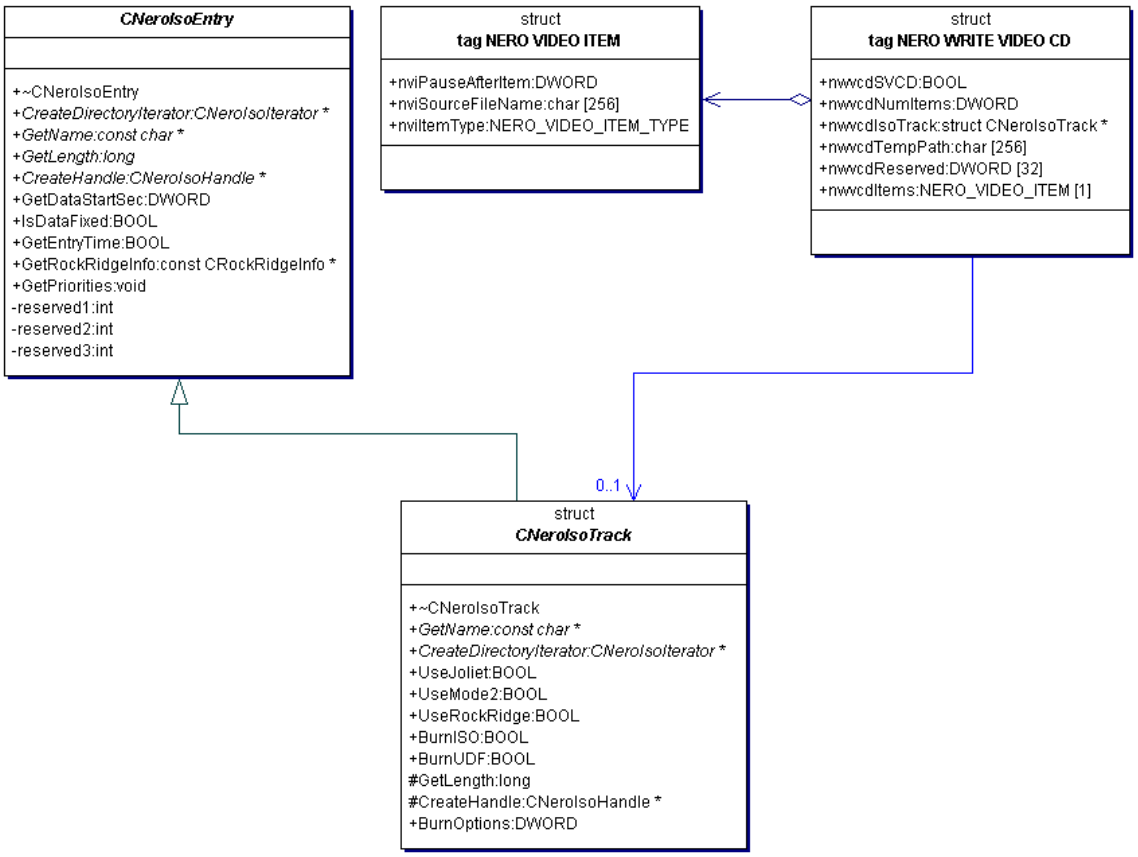
```
typedef struct tag_NERO_WRITE_IMAGE
{
  char nwiImageFileName[252];
  const char *nwiLongImageFileName;
  NERO_MEDIA_TYPE nwiMediaType;
  DWORD nwiReserved[32];
} NERO_WRITE_IMAGE;
```

Description of structure members	
nwiImageFileName	Deprecated, use nwiLongImageFileName instead.
nwiLongImageFileName	Name of the NRG file to burn.
nwiMediaType	Media on which the image should be written. If set to MEDIA_NONE the default media type of the image will be used.

Identifier	Introduced in *NeroAPI* version
NERO_WRITE_IMAGE	5.0.3.9
nwiImageFileName	5.5.6.8: ISO and CUE possible 6.0.0.0: Size reduced from 256 to 252
nwiLongImageFileName	6.0.0.0
nwiMediaType	6.3.0.6

7.1.65. NERO_WRITE_VIDEO_CD

NERO_WRITE_VIDEO_CD is passed to the NeroBurn function in the pWriteCD parameter, when burning video content.




```
typedef struct tag_NERO_WRITE_VIDEO_CD
{
  BOOL  nwvcdSVCD;
  DWORD  nwvcdNumItems;
  struct CNeroIsoTrack  *nwvcdIsoTrack;
  char nwvcdTempPath[252];
  const char *nwvcdLongTempPath;
#ifdef __cplusplus
      VCDEngine::IVCDFSContentGenerator *(*nwvcdCustomVCDEngine)
                (VCDEngine::IVCDMediaDescription*desc,
                 FileSystemContent::IFileSystemDescContainer *pFSDC);
#else
      void *nwvcdCustomVCDEngine;
#endif
  NERO_VIDEO_RESOLUTION nwvcdEncodingResolution;
  DWORD nwvcdReserved[31];
  NERO_VIDEO_ITEM nwvcdItems[1];
} NERO_WRITE_VIDEO_CD;
```

Description of structure members	
nvcdSVCD	If TRUE, write a SVCD.
nwvcdNumItems	Number of Video/Super Video Items.
nwvcdIsoTrack	Pointer to an ISO Track.
nwvcdTempPath	Deprecated, use nwvcdLongTempPath instead.
nwvcdLongTempPath	Where the encoded files will be temporary stored.
nwvcdCustomVCDEngine	
nwvcdEncodingResolution	Select the encoding resolution for the video. This option only has effects for video items of type NERO_NONENCODED_VIDEO_ITEM.
nwvcdReserved	Should be zero.
nwvcdItems	List of Video/Super Video Items.

Identifier	Introduced in *NeroAPI* version
NERO_WRITE_VIDEO_CD	5.0.3.9
nwvcdTempPath	5.5.5.3
nwvcdCustomVCDEngine	5.5.7.6
nwvcdTempPath	5.5.5.3: Size reduced from 256 to 252.
nwvcdLongTempPath	5.5.5.3
nwvcdEncodingResolution	6.0.0.17

7.1.66. NERO_UDF_REVISION

UDF revision type, used by the NeroSetDesiredUDFRevision function.

```
typedef enum
{
  NUR_UDF_INVALID,
  NUR_UDF_102,
  NUR_UDF_150,
  NUR_UDF_200,
  NUR_UDF_201,
  NUR_UDF_250
} NERO_UDF_REVISION;
```

Identifier	Introduced in *NeroAPI* version
NUR_UDF_250	6.6.0.13

7.1.67. NERO_UDF_PARTITION_TYPE

UDF partition type, used by the NeroSetDesiredUDFPartitionType function.

```
typedef enum
{
  NUPT_INVALID,
  NUPT_PHYSICAL,
  NUPT_VIRTUAL,
```

```
  NUPT_SPARING

} NERO_UDF_PARTITION_TYPE;
```

7.1.68. NEROAPI_BURN_ERROR

This is the return type of the NeroBurn function. It indicates whether the burn process was successful or not, and provides a reason if it failed.

```
typedef enum
{
      NEROAPI_BURN_OK=0,
      NEROAPI_BURN_UNKNOWN_CD_FORMAT,
      NEROAPI_BURN_INVALID_DRIVE,
      NEROAPI_BURN_FAILED,
      NEROAPI_BURN_FUNCTION_NOT_ALLOWED,
      NEROAPI_BURN_DRIVE_NOT_ALLOWED,
      NEROAPI_BURN_USER_ABORT,
      NEROAPI_BURN_BAD_MESSAGE_FILE
} NEROAPI_BURN_ERROR;
```

Identifier	Introduced in *NeroAPI* version
NEROAPI_BURN_ERROR	5.0.3.9
NEROAPI_BURN_BAD_MESSAGE_FILE	6.0.0.0

7.1.69. NEROAPI_OPTION

Possible global *Nero* options. Used when calling NeroSetOption.

```
typedef enum
{
  NEROAPI_OPTION_MSG_FILE_NAME,
  NEROAPI_OPTION_WRITE_BUFFER_SIZE ,
  NEROAPI_OPTION_USER_DLG_CALLBACK,
  NEROAPI_OPTION_IDLE_CALLBACK,
  NEROAPI_OPTION_OVERBURN,
  NEROAPI_OPTION_DVDOVERBURN,
  NEROAPI_OPTION_ALLOW_16BITUNICODE_DVDVIDEO_VOLUMELABEL
} NEROAPI_OPTION;
```

Description of enumerators	
NEROAPI_OPTION_MSG_FILE_NAME	Used for changing the file name for the Nero error messages.
NEROAPI_OPTION_WRITE_BUFFER_SIZE	Set write buffer size. Value points onto an integer containing the size in byte.
NEROAPI_OPTION_USER_DLG_ CALLBACK	Set the user dialog callback, overwriting nstUserDialog of the settings structure passed to NeroInit. Pass a pointer to a NERO_CALLBACK structure

Description of enumerators	
	as value. After returning, the struct will contain the previous user callback.
NEROAPI_OPTION_IDLE_CALLBACK	Set the idle callback, overwriting nstIdle of the settings structure passed to NeroInit. Pass a pointer to a NERO_CALLBACK structure as value. After returning, the struct will contain the previous idle callback.
NEROAPI_OPTION_OVERBURN	Enable/Disable overburning. Value points to a DWORD containing the overburn size in blocks. If the value is NULL, disable overburning.
NEROAPI_OPTION_DVDOVERBURN	Enable/Disable overburning for DVDs. The value is a pointer to a DWORD containing the overburn size (the total size of the DVD) in blocks. If the value is NULL, disable overburning for DVDs.
NEROAPI_OPTION_ALLOW_ 16BITUNICODE_DVDVIDEO_ VOLUMELABEL	Enable/Disable volume labels burned as 16 bit unicode for DVD-Video compatible compilations. Normally, this violates the DVD-Video specification. However, some applications might want to enable it anyway. If this option is disabled, NeroAPI will convert characters that are not DVD-Video compatible to '_'. If the value is NULL, the 16 bit unicode volume label feature is disabled, otherwise enabled.

Identifier	Introduced in *NeroAPI* version
NEROAPI_OPTION	5.0.3.9
NEROAPI_OPTION_WRITE_BUFFER_SIZE	5.5.5.0
NEROAPI_OPTION_USER_DLG_CALLBACK	6.0.0.0
NEROAPI_OPTION_IDLE_CALLBACK	6.0.0.0
NEROAPI_OPTION_OVERBURN	6.0.0.27
NEROAPI_OPTION_DVDOVERBURN	6.3.1.12
NEROAPI_OPTION_ALLOW_16BITUNICODE_ DVDVIDEO_VOLUMELABEL	6.6.0.4

7.1.70. NEROAPI_INIT_ERROR

Used when informing the user about the result of a call to NeroInit. Provides some additional information in case of failed initialization.

```
typedef enum
{
  NEROAPI_INIT_OK=0,
  NEROAPI_INIT_INVALID_ARGS,
  NEROAPI_INIT_UNSPECIFIED_ERROR,
  NEROAPI_INIT_INVALID_SERIAL_NUM,
  NEROAPI_INIT_DEMOVERSION_EXPIRED,
  NEROAPI_INIT_ALREADY_INITIALISED,
  NEROAPI_INIT_CANNOT_LOCK
} NEROAPI_INIT_ERROR;
```

Identifier	Introduced in *NeroAPI* version
NERO_INIT_ERROR	5.0.3.9
NEROAPI_INIT_DEMOVERSION_EXPIRED	5.5.1.1
NEROAPI_INIT_ALREADY_INITIALISED	5.5.2.4
NEROAPI_INIT_CANNOT_LOCK	5.5.5.2

7.1.71. NEROAPI_SCSI_DEVTYPE

Code to scan the SCSI/IDE bus and get information about the available WORM/CD-ROM drives.

```
typedef enum
{
  NEA_SCSI_DEVTYPE_UNKNOWN,
  NEA_SCSI_DEVTYPE_WORM,
  NEA_SCSI_DEVTYPE_CDROM,
  NEA_SCSI_DEVTYPE_UNSUPPORTED_WORM
} NEROAPI_SCSI_DEVTYPE;
```

Description of enumerators	
NEA_SCSI_DEVTYPE_UNKNOWN	Type information not available
NEA_SCSI_DEVTYPE_WORM	Write once. A CD-burner.
NEA_SCSI_DEVTYPE_CDROM	Read only. A CD-ROM drive.
NEA_SCSI_DEVTYPE_ UNSUPPORTED_WORM	Can write but is not supported by the *NeroAPI*.

Identifier	Introduced in *NeroAPI* version
NEROAPI_SCSI_DEVTYPE	5.0.3.9
NEA_SCSI_DEVTYPE_UNSUPPORTED_WORM	5.5.6.5

7.1.72. NERODLG_ICON_TYPE

Used by the NERODLG_MESSAGEBOX structure.

```
typedef enum
{
      NDIT_INFO = 0,
      NDIT_WARNING = 1,
      NDIT_ERROR = 2,
      NDIT_QUESTION =3
} NERODLG_ICON_TYPE;
```

Description of enumerators	
NDIT_INFO	An info icon.
NDIT_WARNING	A warning icon.
NDIT_ERROR	An error icon.
NDIT_QUESTION	A question icon.

Identifier	Introduced in *NeroAPI* version
NERODLG_ICON_TYPE	6.0.0.6

7.1.73. NERODLG_MESSAGE_TYPE

```
typedef enum
{
      NDMT_OK = 0,
      NDMT_YESNO = 1,
      NDMT_OKCANCEL = 2,
      NDMT_RETRYCANCEL = 3,
      NDMT_ABORTRETRYIGNORE = 4,
      NDMT_YESNOCANCEL = 5
} NERODLG_MESSAGE_TYPE;
```

Description of enumerators	
NDMT_OK	An info dialog with only an OK button. The return value is ignored.
NDMT_YESNO	A dialog with a yes and a no button. Return DLG_RETURN_YES for Yes and DLG_RETURN_NO for No.
NDMT_OKCANCEL	A dialog with an OK and a Cancel button. Return DLG_RETURN_OK for OK and DLG_RETURN_CANCEL for Cancel.
NDMT_RETRYCANCEL	A dialog with a retry and a cancel button. Return DLG_RETURN_RETRY for Retry and DLG_RETURN_CANCEL for Cancel.

Description of enumerators	
NDMT_ABORTRETRYIGNORE	A dialog with an abort, a retry and a ignore button. Return DLG_RETURN_IGNORE for Ignore, DLG_RETURN_RETRY for Retry and DLG_RETURN_ABORT for Abort.
NDMT_YESNOCANCEL	A dialog with a yes, a no and a cancel button. Return DLG_RETURN_YES for Yes, DLG_RETURN_NO for No and DLG_RETURN_CANCEL for Cancel.

Identifier	Introduced in *NeroAPI* version
NERODLG_MESSAGE_TYPE	6.0.0.6

7.1.74. NERODLG_MESSAGEBOX

Used to describe a custom message box dialog. The data pointer will point to this structure when a DLG_MESSAGEBOX user callback occurs.

```
typedef struct
{
      NERODLG_MESSAGE_TYPE type;
      NERODLG_ICON_TYPE icon;
      NeroUserDlgInOut defaultReturn;
      const char* message;
} NERODLG_MESSAGEBOX;
```

Description of structure members	
type	The type of the message, see DLG_MESSAGE_TYPE.
icon	The icon for the message, see DLG_ICON_TYPE.
defaultReturn	The default return value.
message	The message to display.

Identifier	Introduced in *NeroAPI* version
NERODLG_MESSAGEBOX	6.0.0.6

7.1.75. NeroUserDlgInOutEnum

Ask how to proceed by offering the user some choices.

Due to historical reasons, some of the enumerators refer to CD media.

Please bear in mind that the values can be used with DVD media as well!

```
typedef enum NeroUserDlgInOutEnum {
      DLG_RETURN_OK = -8,
      DLG_RETURN_YES = -7,
      DLG_RETURN_RETRY = -6,
      DLG_RETURN_IGNORE = -5,
      DLG_RETURN_NO = -4,
      DLG_RETURN_CANCEL = -3,
```

```
      DLG_RETURN_ABORT = -2,

      DLG_RETURN_NOT_HANDLED = -1,

      DLG_RETURN_EXIT = 0,
      DLG_RETURN_FALSE = 0,
      DLG_RETURN_TRUE = 1,
      DLG_DISCONNECT = 2,
      DLG_RETURN_ON_RESTART = 3,
      DLG_RETURN_RESTART = 4,
      DLG_RETURN_CONTINUE = 5,
      DLG_DISCONNECT_RESTART = 6,
      DLG_AUTO_INSERT = 7,
      DLG_RETURN_INSTALL_DRIVER = 8,
      DLG_RETURN_OFF_RESTART = 9,
      DLG_RESTART = 10,
      DLG_AUTO_INSERT_RESTART = 11,
      DLG_SETTINGS_RESTART = 12,
      DLG_OVERBURN = 13,
      DLG_AUDIO_PROBLEMS = 14,
      DLG_WAITCD = 15,
      DLG_WAITCD_REMINDER = 16,
      DLG_WAITCD_DONE = 17,
      DLG_COPY_QUALITY_LOSS = 18,
      DLG_COPY_FULLRISK = 19,
      DLG_FILESEL_IMAGE = 20,
      DLG_BURNIMAGE_CANCEL = 21,
      DLG_NON_EMPTY_CDRW = 22,
      DLG_COMP_REC_CONFLICT = 23,
      DLG_WRONG_MEDIUM = 24,
      DLG_ROBO_MOVECD = 25,
      DLG_ROBO_MOVECD_DONE = 26,
      DLG_ROBO_USERMESSAGE = 27,
      DLG_WAITCD_MEDIA_INFO = 28,
      DLG_MESSAGEBOX = 29,
      DLG_MAX
} NeroUserDlgInOut;
```

Description of enumerators	
DLG_RETURN_OK	Return code for DLG_MESSAGEBOX: OK.
DLG_RETURN_YES	Return code for DLG_MESSAGEBOX: Yes.
DLG_RETURN_RETRY	Return code for DLG_MESSAGEBOX: Retry.
DLG_RETURN_IGNORE	Return code for DLG_MESSAGEBOX: Ignore.
DLG_RETURN_NO	Return code for DLG_MESSAGEBOX: No.
DLG_RETURN_CANCEL	Return code for DLG_MESSAGEBOX: Cancel.
DLG_RETURN_ABORT	Return code for DLG_MESSAGEBOX: Abort.
DLG_RETURN_NOT_HANDLED	Return this if an enum is not handled by the callback.

Description of enumerators	
DLG_RETURN_EXIT	Exit application / stop writing.
DLG_RETURN_FALSE	False.
DLG_RETURN_TRUE	True.
DLG_DISCONNECT	"Disconnect is turned off in the system configuration. This may cause serious problems while burning: your disc might be damaged, or the system might hang up."
DLG_RETURN_ON_RESTART	Turn on disconnect and restart windows.
DLG_RETURN_RESTART	Do not change the disconnect option and restart windows.
DLG_RETURN_CONTINUE	Continue at your own risk. (Use DLG_RETURN_EXIT instead to terminate the process.)
DLG_DISCONNECT_RESTART	Same as DLG_DISCONNECT, but restarting has been selected already and must not be canceled, so valid return codes are only DLG_RETURN_ON_RESTART and DLG_RETURN_RESTART.
DLG_AUTO_INSERT	"Auto Insert Notification is turned on in the system configuration. This may cause serious problems while burning: your disc might be damaged, or the system might hang up. Nero is able to burn discs with Auto Insert Notification turned on if all necessary drivers are installed."
DLG_RETURN_INSTALL_DRIVER	Install IO driver which temporarily disables auto insert. Note: this only works if the additional argument for the callback is not NULL, otherwise it should not be offered to the user.
DLG_RETURN_OFF_RESTART	Change autoinsert and restart Windows.
DLG_AUTO_INSERT_RESTART	"Auto Insert Notification is now OFF. You should restart Windows." (displayed after rebooting within program failed and user has to do it manually). The return code is irrelevant.
DLG_SETTINGS_RESTART	"Nero detected some modifications of your PC system configuration and needs to modify some settings. Please restart your PC to make the changes become effective." Allowed return values: DLG_RETURN_RESTART DLG_RETURN_CONTINUE
DLG_OVERBURN	"Sorry, this compilation contains too much data to fit on the disc with respect to the normal disc capacity. Do you want to try overburn writing at your own risk (this might cause read errors at the end of the disc or might even damage your recorder)?" Note: It is also possible, that SCSI/Atapi errors

Description of enumerators	
	occur at the end of the simulation or burning. Even in this case there is a certain chance, that the disc is readable. Allowed return values: DLG_RETURN_TRUE DLG_RETURN_FALSE
DLG_AUDIO_PROBLEMS	The tracks cannot be written as requested. A detailed description of the problem is found in the "data" parameter. It is a DWORD with bits set according to the AUP (Audio Problem) constants. Return DLG_RETURN_TRUE to fix the problems by adapting the track settings. Return DLG_RETURN_FALSE to stop writing.
DLG_WAITCD	This dialog type differs slightly from the other ones: It should pop up a message and return immediately while still showing the message, so that the API can test for the expected disc in the meantime. During this time, the NERO_IDLE_CALLBACK will be called to give the application a chance to update its display and to test for user abort. The API might call DLG_WAITCD several times to change the text. The text depends on the "data" argument that is passed to the NERO_USER_DIALOG callback. It is the enumeration NERO_WAITCD_TYPE specified below.
DLG_WAITCD_REMINDER	It is time to remind the user of inserting the disc: play a jingle, flash the screen, etc. Called only once after a certain amount of time of no disc being inserted.
DLG_WAITCD_DONE	Close the message box again, we are done.
DLG_COPY_QUALITY_LOSS	Tell the user that there will be quality loss during the copy and ask if he wants to continue anyway.
DLG_COPY_FULLRISK	PROCEED AT YOUR OWN RISK message.
DLG_FILESEL_IMAGE	Ask the user the path of the file which will be generated by the Image Recorder. The "data" argument points on a 256 bytes buffer that has to be filled with the image path. Returning DLG_RETURN_EXIT will stop the burn process.
DLG_BURNIMAGE_CANCEL	Tell that there is not enough space on disk to produce this image.
DLG_NON_EMPTY_CDRW	Tell the user that the rewritable media is not empty. Starting from *NeroAPI* 5.5.3.0, the "data" argument contains the device handle of the

Description of enumerators	
	recorder. It will only be called if the NBF_DETECT_EMPTY_CDRW flag is given to the NeroBurn function. Returning DLG_RETURN_EXIT will stop the burn process. Returning DLG_RETURN_CONTINUE will continue the burn process. Returning DLG_RETURN_RESTART will ask the user for another disc.
DLG_COMP_REC_CONFLICT	Tell the user that the compilation cannot be written on that particular recorder and that the user should modify his compilation settings or burn the disc on another recorder, which supports the required medium type.
DLG_WRONG_MEDIUM	Another type of medium must be used to burn this compilation.
DLG_ROBO_MOVECD	Implementation of the DLG_ROBO_MOVECD dialog types must behave like the DLG_WAITCD type, that is, operate in a non-blocking way. The data structure passed to this callback is specified as * ROBOMOVEMESSAGE below.
DLG_ROBO_MOVECD_DONE	Destroy a MoveCD dialog. (void*)data cast to an int will contain the * id of the MoveCD dialog to be removed.
DLG_ROBO_USERMESSAGE	Show dialog message transmitted by the robot driver. Must return one of the constants below. The data structure passed as the data pointer is specified as ROBOUSERMESSAGE below. Return DLG_RETURN_FALSE or DLG_RETURN_TRUE.
DLG_WAITCD_MEDIA_INFO	Provide information about which media is expected and which media is currently present in the recorder. The data pointer passed is a pointer on the NERO_DLG_WAITCD_MEDIA_INFO structure. The value returned is ignored.
DLG_MESSAGEBOX	Open a custom message box dialog. The type and the message of the dialog are described with a struct NERODLG_MESSAGEBOX which is given as data pointer. See comments for NERODLG_MESSAGE_TYPE which values to return.
DLG_MAX	Not used.

Identifier	Introduced in *NeroAPI* version
DLG_COMP_REC_CONFLICT	5.5.3.2
DLG_WRONG_MEDIUM	5.5.3.2
DLG_OVERBURN	6.0.0.27: The data parameter is a pointer to struct DLG_OVERBURN_INFO.
DLG_RETURN_OK	6.0.0.6
DLG_RETURN_YES	6.0.0.6
DLG_RETURN_RETRY	6.0.0.6
DLG_RETURN_IGNORE	6.0.0.6
DLG_RETURN_NO	6.0.0.6
DLG_RETURN_CANCEL	6.0.0.6
DLG_RETURN_ABORT	6.0.0.6
DLG_RETURN_NOT_HANDLED	6.0.0.6
DLG_MESSAGEBOX	6.0.0.6

7.1.76. ROBOMOVEMESSAGE

This struct is used in the context of the NeroUserDlgInOut callback.

```
typedef struct
{
   int id;
   ROBOMOVENODE source;
   ROBOMOVENODE destination;
} ROBOMOVEMESSAGE;
```

Description of structure members	
id	In future versions, we may have more than one robot moving at a time. So this ID identifies the movement action and will be used to remove it with DLG_ROBO_MOVECD_DONE.
source	Source position.
destination	Destination position.

7.1.77. ROBOMOVENODE

Enumeration of node types.

```
typedef enum
{
   RMN_INPUT,
   RMN_RECORDER,
   RMN_OUTPUT,
   RMN_PRINTER,
   RMN_WASTEBIN
} ROBOMOVENODE;
```

7.1.78. ROBOUSERMESSAGE

This struct is used as data parameter when the NeroUserDlgInOut callback is called with the type DLG_ROBO_USERMESSAGE.

```
typedef struct
{
   ROBOUSERMESSAGETYPE message_type;
   const char *message;
} ROBOUSERMESSAGE;
```

Description of structure members	
message_type	The type of message, see ROBOUSERMESSAGETYPE description.
message	Message text.

7.1.79. ROBOUSERMESSAGETYPE

This enum type is used by the ROBOUSERMESSAGE struct.

```
typedef enum
{
   RUMT_ERROR,
   RUMT_WARNING,
   RUMT_QUESTION,
   RUMT_HINT
} ROBOUSERMESSAGETYPE;
```

7.2. Functions

7.2.1. NeroAudioCreateTargetItem

This helper function creates a target item and returns a handle.

```
NEROAPI_API NERO_AUDIO_ITEM_HANDLE NeroAudioCreateTargetItem(int
                                                    iFormatNumber);
```

Description of parameters	
iFormatNumber	The format index number as used with the NeroAudioGetFormatInfo function.

7.2.2. NeroAudioCloseItem

This is a helper function to close an audio target item.

```
NEROAPI_API BOOL  NeroAudioCloseItem(NERO_AUDIO_ITEM_HANDLE hItem);
```

Description of parameters	
hItem	The handle of the item that will be closed.

7.2.3. NeroAudioGetFormatInfo

This helper function retrieves information about audio formats. When it returns false, there are no further formats available.

```
NEROAPI_API BOOL  NeroAudioGetFormatInfo (int iNum,
                                          NERO_AUDIO_FORMAT_INFO *pFI);
```

Description of parameters	
iNum	Format index number, use 0 to retrieve the first available format.
pFI	Pass a pointer to a NERO_AUDIO_FORMAT_INFO object; it will be filled with information about the format.

7.2.4. NeroAudioGUIConfigureItem

This function will open a configuration dialog for audio items. Instead of phItem, a value of NULL can be passed to configure the whole plug-in manager.

NeroAudioGUIConfigureItem can only be used from GUI applications.

```
NEROAPI_API NERO_CONFIG_RESULT NeroAudioGUIConfigureItem(
                               NERO_AUDIO_ITEM_HANDLE *phItem, int iNum);
```

Description of parameters	
phItem	An array of handles that belong to configurable items.
iNum	The number of configurable items in phItem.

7.2.5. NeroBurn

Burns a media.

```
NEROAPI_BURN_ERROR NADLL_ATTR NeroBurn
(
 NERO_DEVICEHANDLE  aDeviceHandle,
 NERO_CD_FORMAT  CDFormat,
 const void*  pWriteCD,
 DWORD  dwFlags,
 DWORD  dwSpeed,
 NERO_PROGRESS*  pNeroProgress
);
```

Description of parameters		
pwriteCD	Must point on a NERO_WRITE_CD, NERO_CD_COPY or a NERO_WRITE_VIDEO_CD structure.	
dwFlags	Some options for burning:	
	NBF_SPEED_TEST	Test speed of source first.
	NBF_SIMULATE	Simulate writing before actually writing.
	NBF_WRITE	Really write at the end.
	NBF_DAO	Write in DAO.
	NBF_CLOSE_SESSION	Only close the session, not the whole disc.
	NBF_CD_TEXT	Write CD text - will be ignored if not supported by drive.
	NBF_BURN_PROOF	Present for compatibility: will enable any buffer underrun protection feature even if it is not "burn proof"
	NBF_BUF_UNDERRUN_PROT	Enable safer burn mode.
	NBF_DISABLE_ABORT	The abort callback will be called.
	NBF_DETECT_NON_EMPTY_CDRW	The DLG_NON_EMPTY_CDRW user callback will be called when trying to burn onto a non empty CDRW.
	NBF_DISABLE_EJECT	CD will not be ejected at the end of the burn process.
	NBF_VERIFY	Verify file system after writing. Works for ISO only.
	NBF_SPEED_IN_KBS	Interpret the dwSpeed as KB/s instead of multiple of 150 KB/s.
	NBF_DVDP_BURN_30MM_AT_LEAST	DVD+R/RW high compatibility mode (at least 1GB will be written)
	NBF_CD_TEXT_IS_JAPANESE	If NBF_CD_TEXT and NBF_CD_TEXT_IS_JAPANESE are set, then the CD Text is treated as Japanese CD Text.
	NBF_BOOKTYPE_DVDROM	If NBF_BOOKTYPE_DVDROM is set, the booktype of a burned DVD will be set to DVDROM

Description of parameters		
	NBF_NO_BOOKTYPE_ CHANGE	Do not change the booktype of DVD, even if the NeroAPI default setting is to change the booktype to DVD-ROM.
	NBF_RESERVED3	Reserved.
	NBF_RESERVED2	Reserved.
	NBF_RESERVED	Reserved.
dwSpeed	In KB/s if NBF_SPEED_IN_KBS is present, in multiple of 150 KB/s otherwise.	

Identifier	Introduced in *NeroAPI* version
NeroBurn	5.0.3.9
NBF_DISABLE_EJECT	5.5.1.1
NBF_SPEED_IN_KBS	5.5.5.5
NBF_DVDP_BURN_30MM_AT_LEAST	5.5.8.0
NBF_CD_TEXT_IS_JAPANESE	5.5.9.17
NBF_BOOKTYPE_DVDROM	5.5.10.7
NBF_NO_BOOKTYPE_CHANGE	6.0.0.24

7.2.6. NeroClearErrors

Clear errors and log (done automatically for every read or write function, but can be used to avoid false memory leak warnings).

```
NEROAPI_API void NADLL_ATTR NeroClearErrors ();
```

Identifier	Introduced in *NeroAPI* version
NeroClearErrors	5.0.3.9

7.2.7. NeroCloseDevice

Close a device.

```
NEROAPI_API void NADLL_ATTR NeroCloseDevice(NERO_DEVICEHANDLE
aDeviceHandle);
```

Identifier	Introduced in *NeroAPI* version
NeroCloseDevice	5.0.3.9

7.2.8. NeroConvertToValidFileSystemName

Converts a given filename/volume label to a valid filename for a certain file system. The function returns TRUE, if the filename was converted, FALSE otherwise.

If out is NULL and the function returns TRUE, the filename requires conversion.

```
NEROAPI_API  BOOL NADLL_ATTR NeroConvertToValidFileSystemName(
                                        const WCHAR *in,
                                        WCHAR *out,
                                        unsigned long outBufferSize,
                                        NERO_FS_NAMETYPE nameType,
                                        NERO_FS_TYPE dwFilesystemFlag,
                                        void *reserved);
```

Description of parameters	
in	The filename/volume label to convert.
out	The buffer that will receive the converted filename.
outBufferSize	The out buffer size in bytes.
nameType	The type of the name (filename or volume label).
dwFilesystem Flag	Target file system, for which the name is converted.
reserved	Reserved.

Identifier	Introduced in *NeroAPI* version
NeroConvertToValidFileSystemName	6.6.0.4

7.2.9. NeroCopyIsoItem

Create a copy of an existing NERO_ISO_ITEM object.

This is a safe way to obtain an exact copy of NERO_ISO_ITEM objects imported from a previous session. Note that the new NERO_ISO_ITEM's extItem, userData and subDirFirstItem members are set to NULL.

longFilename will only be copied if this item is a reference. In that case longFilename had been allocated by the *NeroAPI*.

```
NEROAPI_API NERO_ISO_ITEM * NADLL_ATTR
NeroCopyIsoItem (const NERO_ISO_ITEM *iso_item);
```

Identifier	Introduced in *NeroAPI* version
NeroCopyIsoItem	5.5.9.9

7.2.10. NeroCreateIsoItem

Macro for automatically filling the size_t member of NeroCreateIsoItemOfSize.

```
#define NeroCreateIsoItem()  NeroCreateIsoItemOfSize(sizeof(struct
NERO_ISO_ITEM))
```

Identifier	Introduced in *NeroAPI* version
NeroCreateIsoItem	5.0.3.9

7.2.11. NeroCreateIsoItemOfSize

Allocates an instance of the NERO_ISO_ITEM structure of size size_t.

```
NEROAPI_API struct NERO_ISO_ITEM * NADLL_ATTR
NeroCreateIsoItemOfSize(size_t);
```

Identifier	Introduced in *NeroAPI* version
NeroCreateIsoItemOfSize	5.0.3.9

7.2.12. NeroCreateIsoTrackEx

Create an ISO track from a NERO_ISO_ITEM tree.

```
NEROAPI_API struct CNeroIsoTrack * NADLL_ATTR NeroCreateIsoTrackEx(
                                        struct NERO_ISO_ITEM *root,
                                        const char *name,
                                        DWORD flags);
```

Description of parameters		
root	First item of the root directory.	
name	Name of the CD.	
flags	available constants:	
	NCITEF_USE_JOLIET	(1<<0) Create a Joliet Track.
	NCITEF_USE_MODE2	(1<<1) Create a Mode 2 Track.
	NCITEF_USE_ROCKRIDGE	(1<<2) Create a RockRidge Track.
	NCITEF_CREATE_ISO_FS	(1<<3) Create an ISO File System Track.
	NCITEF_CREATE_UDF_FS	(1<<4) Create a Universal Disk Format File System Track.
	NCITEF_CREATE_HFS_FS	(1<<5) Not yet available.

Description of parameters		
	NCITEF_DVDVIDEO_ REALLOC	(1<<6) Perform reallocation of files in the VIDEO_TS directory. Beginning with *NeroAPI* 6.3.1.4: Also create layer break if writing on a double layer media.
	NCITEF_USE_STRUCT	(1<<7) 'name' points to an argument struct instead of name. For special needs you have to give a pointer to NeroCITEArgs instead of a name, e.g. when burning a CD with two different file systems. Set this flag to tell NeroCreateIsoTrackEx that the name is a NeroCITEArgs struct and set the flags for the burn options with NeroCITEArgs::dwBurnOptions. ‘root’ should also be NULL in this case.
	NCITEF_RESERVED1	(1<<8) Reserved for future use.
	NCITEF_USE_ALLSPACE	(1<<9) Use all space available on the medium for the volume to be created. Supported for DVD+-RW only.
	NCITEF_RESERVED2	(1<<10) Reserved for future use.
	NCITEF_RESERVED3	(1<<11) Reserved for future use.
	NCITEF_RESERVED4	(1<<12) Reserved for future use.
	NCITEF_RELAX_JOLIET	(1<<13) Relax Joliet filename length limitations and allow a maximum of 109 characters per filename.
	NCITEF_DVDVIDEO_CMPT	(1<<14) Create DVD-Video compatible medium. NCITEF_CREATE_ISO_FS and NCITEF_CREATE_UDF_FS must be set. NCITEF_DVDVIDEO_REALLOC may be set to reallocate DVD-Video .IFO pointers. **Note:** *NeroAPI* versions prior or equal to 6.0.0.13 will implicitly enable DVD-Video compatibility when DVD-Video content is found within the compilation.
	NCITEF_RESERVED5	(1<<15) Reserved for future use.
	NCITEF_RESERVED6	(1<<16) Reserved for future use.
	NCITEF_RESERVED7	(1<<17) Reserved for future use.

Identifier	Introduced in *NeroAPI* version
NeroCreateIsoTrackEx	5.0.3.9
NCITEF_DVDVIDEO_REALLOC	5.5.7.8
NCITEF_USE_STRUCT	5.5.9.0
NCITEF_USE_ALLSPACE	5.5.9.17
NCITEF_RELAX_JOLIET	5.9.10.17
NCITEF_DVDVIDEO_CMPT	6.0.0.13
NCITEF_RESERVED5	6.0.0.13

7.2.13. NeroCreateProgress

Creates a correctly initialized NERO_PROGRESS structure.

The memory used by the structure must be freed with NeroFreeMem when no longer needed.

```
NEROAPI_API NERO_PROGRESS* NADLL_ATTR NeroCreateProgress();
```

Identifier	Introduced in *NeroAPI* version
NeroCreateProgress	6.0.0.0

7.2.14. NeroDAE

Digital Audio Extraction. Aborting will not be reported by NeroGetLastError. Incomplete target files are not deleted.

```
int NADLL_ATTR NeroDAE
(
 NERO_DEVICEHANDLE  aDeviceHandle,
 DWORD  dwTrackStartBlk,
 DWORD  dwTrackLengthInBlks,
 const  NERO_DATA_EXCHANGE  *pDestDataExchg,
 DWORD  iSpeedInX,
 NERO_CALLBACK*  pNeroProgressCallback
);
```

Description of parameters	
iSpeedInX	Speed of extraction, 0 means maximum speed
pNeroProgressCallback	Has to be a NERO_PROGRESS_CALLBACK.

Identifier	Introduced in *NeroAPI* version
NeroDAE	5.0.3.9

7.2.15. NeroDone

Call this function before closing the DLL. This is necessary because some clean-up actions like stopping threads cannot be done in the close function of the DLL.

NeroDone returns TRUE if some memory blocks were not unallocated using NeroFreeMem. They are dumped in the debug output.

NeroDone returns FALSE if it succeeded.

```
NEROAPI_API BOOL NADLL_ATTR NeroDone ();
```

Identifier	Introduced in *NeroAPI* version
NeroDone	5.0.3.9
Return type changed from void to BOOL.	6.0.0.0

7.2.16. NeroEjectLoadCD

Returns zero if successful or an error code if not. FALSE in parameter “eject” loads a disc, TRUE ejects.

Due to historical reasons, the function name refers to CD media. Please bear in mind that it can be used with DVD media as well!

```
NEROAPI_API int NADLL_ATTR NeroEjectLoadCD(NERO_DEVICEHANDLE
aDeviceHandle,BOOL eject);
```

Identifier	Introduced in *NeroAPI* version
NeroEjectLoadCD	5.0.3.9

7.2.17. NeroEraseCDRW

Erase the loaded CD. With parameter “mode” set to 0 the function will erase the entire CD. If “mode” is set to “1” a quick erase routine will be performed.

This function is deprecated! Please use NeroEraseDisc instead!

```
NEROAPI_API int NADLL_ATTR NeroEraseCDRW(NERO_DEVICEHANDLE DeviceHandle,
                                         int mode);
```

Identifier	Introduced in *NeroAPI* version
NeroEraseCDRW	5.0.3.9

7.2.18. NeroEraseDisc

Erase the disc inserted in the given recorder. This function returns error codes of NERO_DRIVE_ERROR type.

```
NEROAPI_API int NADLL_ATTR NeroEraseDisc(
                                NERO_DEVICEHANDLE aDeviceHandle,
                                NEROAPI_CDRW_ERASE_MODE mode,
                                DWORD dwFlags,
                                DWORD dwSpeedInKBs
                                void *reserved);
```

Description of parameters		
aDeviceHandle	Recorder handle.	
mode	Erase mode.	
flags	available constants :	
	0	Default behavior: Eject if the recorder requires it.
	NEDF_DISABLE_EJECT	CD will not be ejected at the end of the erasing, even if this is recommended for the selected recorder.
	NEDF_EJECT_AFTER_ERASE	Eject disc after erasing, no matter if this is recommended for the recorder or not.
dwSpeedInKBs	Erasing speed in KB/s.	

Identifier	Introduced in *NeroAPI* version
NeroEraseDisc	6.0.0.0
dwSpeedInKBs	6.6.1.0

7.2.19. NeroEstimateTrackSize

Estimate the total size of a track including data and overhead for the file system. The method returns the size in blocks. Use the flags to specify what exactly has to be taken into account for the calculation.

Warning: Depending on the parameters passed, the returned size might only be an estimated value!

```
NEROAPI_API unsigned int NADLL_ATTR NeroEstimateTrackSize(
                              struct CNeroIsoTrack *pIsoTrack,
                              DWORD dwFlags,
                              NERO_FILESYSTEMTRACK_OPTIONS *pOptions);
```

Description of parameters		
pIsoTrack	The iso track for which to calculate the size.	
dwFlags	Combination of flags:	
	NETS_FILESYSTEM_ OVERHEAD	(1<<0) Calculate file system overhead.
	NETS_DATA	(1<<1) Calculate data size.
	NETS_EXACT_SIZE	(1<<2) Calculate exactly. If this option is specified, file system overhead as well as file data are taken into account. The optional fields of the NERO_ESTIMATETRACKSIZE_OPTIONS structure need to be filled out.
pOptions	Pointer to NERO_FILESYSTEMTRACK_OPTIONS structure.	

Identifier	Introduced in *NeroAPI* version
NeroEstimateTrackSize	6.0.0.14
NETS_EXACT_SIZE	6.0.0.21

7.2.20. NeroFreeCDStamp

Free a CD stamp allocated by NeroImportIsoTrackEx.

```
NEROAPI_API void NADLL_ATTR NeroFreeCDStamp(void *pCDStamp);
```

Identifier	Introduced in *NeroAPI* version
NeroFreeCDStamp	5.0.3.9

7.2.21. NeroFreeIsoItem

Free memory that is used by an instance of the NERO_ISO_ITEM structure.

The memory for NERO_ISO_ITEM.longFileName will only be released if NERO_ISO_ITEM.isReference member evaluates to TRUE. This stems from the NeroImportDataTrack behavior, where the *NeroAPI* allocates longFileName.

```
NEROAPI_API void NADLL_ATTR NeroFreeIsoItem(struct NERO_ISO_ITEM *);
```

Identifier	Introduced in *NeroAPI* version
NeroFreeIsoItem	5.0.3.9

7.2.22. NeroFreeIsoTrack

Free an ISO track previously allocated with NeroCreateIsoTrackEx.

```
NEROAPI_API void NADLL_ATTR NeroFreeIsoTrack(struct CNeroIsoTrack
*track);
```

Identifier	Introduced in *NeroAPI* version
NeroFreeIsoTrack	5.0.3.9

7.2.23. NeroFreeIsoItemTree

Free an NERO_ISO_ITEM including all linked items.

It is required that all NERO_ISO_ITEMS in the tree have been created by either the NeroCreateIsoItem or the NeroImportDataTrack function.

The memory for NERO_ISO_ITEM.longFileName will be released if NERO_ISO_ITEM.isReference member evaluates to TRUE. This stems from the NeroImportDataTrack behavior, where the *NeroAPI* allocates memory for longFileName.

```
NEROAPI_API void NADLL_ATTR NeroFreeIsoItemTree(NERO_ISO_ITEM*);
```

Identifier	Introduced in *NeroAPI* version
NeroFreeIsoItemTree	6.0.0.0

7.2.24. NeroFreeMem

The *NeroAPI* never uses static memory. Instead, memory is allocated dynamically on behalf of the application, e.g. for strings. This memory has to be freed with this function. Passing NULL is allowed.

```
NEROAPI_API void NADLL_ATTR NeroFreeMem (void *pMem);
```

Identifier	Introduced in *NeroAPI* version
NeroFreeMem	5.0.3.9

7.2.25. NeroGetAPIVersion

Version management for this API: Returns 1000 for 1.0.0.0

Note: This function is obsolete since NeroAPI 5.5.9.9. Use NeroGetAPIVersionEx instead!

```
NEROAPI_API DWORD NADLL_ATTR NeroGetAPIVersion(void);
```

Identifier	Introduced in *NeroAPI* version
NeroGetAPIVersion	5.0.3.9

7.2.26. NeroGetAPIVersionEx

Fills the pointed numbers (major version high and low, minor version high and low) with the version number and returns true for success. The NeroGetAPIVersion function was extended in *NeroAPI* 5.5.9.9 to support multiple digits. Provide NULL for the “reserved” parameter!

```
NEROAPI_API BOOL NADLL_ATTR NeroGetAPIVersionEx(   WORD *majhi
                                                  ,WORD *majlo
                                                  ,WORD *minhi
                                                  ,WORD *minlo
                                                  ,void *reserved);
```

Identifier	Introduced in *NeroAPI* version
NeroGetAPIVersionEx	5.5.9.9

7.2.27. NeroGetAvailableDrivesEx

Retrieves a list of available WORM and CDROM devices. This list will be freed when calling NeroFreeMem. NeroGetAvailableDrivesEx will return NULL if errors occurred.

The returned information might be inaccurate if another application uses one of the recorders while the identification scan is performed.

Use by another application is indicated by the nsdiDriveError member of NERO_SCSI_DEVICE_INFO being set to NDE_DRIVE_IN_USE.

If the information is inaccurate, it can be updated at a later time by calling NeroUpdateDeviceInfo.

```
NEROAPI_API NERO_SCSI_DEVICE_INFOS * NADLL_ATTR
NeroGetAvailableDrivesEx(NERO_MEDIA_TYPE mediaType, DWORD dwFlags);
```

Description of parameters	
mediaType	Provide speeds values for this kind of media.
dwFlags	Available values: NGADE_GET_LONG_HOST_ADAPTER_NAMES (1<<0) Provides long host adapter names. nsdiLongHostAdapterName should be freed manually for every NERO_SCSI_DEVICE_INFO structure returned.

Identifier	Introduced in *NeroAPI* version
NeroGetAvailableDrivesEx	5.0.3.9
dwFlags	6.6.0.10

7.2.28. NeroGetAvailableSpeeds

Get available write speeds depending on medium type, free with NeroFreeMem. Returns NULL for error.

```
NEROAPI_API NERO_SPEED_INFOS * NADLL_ATTR
NeroGetAvailableSpeeds (NERO_DEVICEHANDLE aDeviceHandle,
                        NERO_ACCESSTYPE accessType,
                        NERO_MEDIA_TYPE mediaType,
                        void *reserved);
```

Identifier	Introduced in *NeroAPI* version
NeroGetAvailableSpeeds	5.5.9.10

7.2.29. NeroGetCDInfo

Due to historical reasons, the function name refers to CD media. Please bear in mind that it can be used with DVD media as well!

Retrieve a pointer to a NERO_CD_INFO structure for the specified device. The allocated memory for the structure has to be freed by using NeroFreeMem. NULL will be returned if an error occurred.

Note: When queried about DVD+RW media, most recorders will return the total capacity of the media, even if parts of it already contain data.

Therefore, when dealing with DVD+RW media, NeroGetVMSInfo should be called after NeroGetCDInfo. NeroGetVMSInfo returns a pointer to a NERO_VMS_INFO structure. This structure contains the next writeable address in the “nvmsiNextWriteableAddress” member. This information can be utilized to determine what portion of the media already is in use.

This strategy will only work if the media has been created as multisession media by *Nero* or the *NeroAPI*. If the media is of non-multisession type, currently there is no way of determining the size of the used portion.

```
NEROAPI_API NERO_CD_INFO * NADLL_ATTR NeroGetCDInfo
(
  NERO_DEVICEHANDLE aDeviceHandle,
  DWORD dwFlags
);
```

Description of parameters		
aDeviceHandle	Device Handle	
dwFlags	available constants :	
	NGCDI_READ_CD_TEXT	(1<<0)
	NGCDI_READ_ISRC	(1<<1) International Standard Recording Code
	NGCDI_USE_HDB	(1<<2) If the recorder and the current media support HD-BURN, give the capacity and the unused blocks for the HD-BURN mode. Note, that if the media is already written in HD-BURN mode, this flag is not necessary.
	NGCDI_EXTENDED	(1<<3) Get extended disc info, e.g. file system type of particular track. This can take a little bit more time, therefore set it only if such information is really required.

Identifier	Introduced in *NeroAPI* version
NeroGetCDInfo	5.0.3.9
NGCDI_READ_ISRC	5.5.8.4
NGCDI_USE_HDB	6.0.0.25
NGCDI_EXTENDED	6.6.0.3

7.2.30. NeroGetCDRWErasingTime

This function is deprecated. Please use NeroGetDiscErasingTime instead!

Returns estimated blanking time for loaded RW media in seconds.

Due to historical reasons, the function name refers to CD media. Please bear in mind that it can be used with DVD media as well!

```
NEROAPI_API int NADLL_ATTR NeroGetCDRWErasingTime(NERO_DEVICEHANDLE
aDeviceHandle,int mode);
```

Description of return values	
-1	No CD inserted.
-2	Recorder does not support CDRW.
-3	The inserted media is not rewriteable.

Identifier	Introduced in *NeroAPI* version
NeroGetCDRWErasingTime	5.0.3.9
Return value “–3”	5.5.7.4

7.2.31. NeroGetCurrentMediumType

NeroGetCurrentMediumType determines the medium type of the disc currently inside the recorder. The function returns MEDIA_NONE if no medium is inside.

```
NEROAPI_API NERO_MEDIA_TYPE NADLL_ATTR NeroGetCurrentMediumType(
                                    NERO_DEVICEHANDLE aDeviceHandle);
```

Description of parameters	
aDeviceHandle	Device Handle.

Identifier	Introduced in *NeroAPI* version
NeroGetCurrentMediumType	6.6.1.0

7.2.32. NeroGetDeviceOption

Get information about a special low level option from a device, e.g. if a device is capable of changing the booktype of a DVD. The returned value must be freed with NeroFreeMem by the caller.

If the option is not available, NULL is returned. The return type depends on the queried option, for example NERO_DEVICEOPTION_BOOKTYPE_DVDROM will make the returned type BOOL*.

```
NEROAPI_API void* NADLL_ATTR NeroGetDeviceOption(
                                    NERO_DEVICEHANDLE aDeviceHandle,
                                    NERO_DEVICEOPTION aOption,
                                    void* reserved);
```

Description of parameters	
aDeviceHandle	Device Handle.
aOption	A device option, e.g. setting the booktype.
reserved	Reserved for future use.

Identifier	Introduced in *NeroAPI* version
NeroGetDeviceOption	5.5.10.7

7.2.33. NeroGetDiscErasingTime

Get rewritable disc erasing time corresponding to the erasing speed in KB/s.

```
NEROAPI_API int NADLL_ATTR NeroGetDiscErasingTime(
                                    NERO_DEVICEHANDLE aDeviceHandle,
                                    NEROAPI_CDRW_ERASE_MODE mode,
                                    DWORD dwSpeedInKBs);
```

Identifier	Introduced in *NeroAPI* version
NeroGetDiscErasingTime	6.6.1.0

7.2.34. NeroGetDiscImageInfo

Get information about a disc image. The result must be released using NeroFreeMem.
In case of an error, NULL is returned.

```
NEROAPI_API NERO_CD_INFO * NADLL_ATTR NeroGetDiscImageInfo(
                              const char *imagePath, void *reserved);
```

Description of parameters	
imagePath	Path to the image file.
reserved	Reserved for future use.

Identifier	Introduced in *NeroAPI* version
NeroGetDiscImageInfo	5.5.9.16

7.2.35. NeroGetErrorLog

All functions returning a DWORD will return zero for success and an error number otherwise. These error numbers are opaque and neither can nor should be interpreted by the application. Instead, localized strings are provided for errors and informative displays. The *NeroAPI* keeps a log of such informative messages or errors.

In case of an error, NeroGetLastError will return more information about the last error and NeroGetErrorLog will show all recorded events.

Both functions return NULL if no error is available. Memory is allocated for the string, which has to be freed with NeroFreeMem.

Note: NeroCloseDrive has to throw away all errors, because they might be bound to the driver. Handle errors before calling it!

```
NEROAPI_API char * NADLL_ATTR NeroGetErrorLog ();
```

Identifier	Introduced in *NeroAPI* version
NeroGetErrorLog	5.0.3.9

7.2.36. NeroGetLastDriveError

Get the last error occurred during communication with a drive.

The following methods set this error value:

- NeroGetCDInfo
- NeroImportDataTrack
- NeroEjectLoadCD
- NeroGetCDRWErasingTime
- NeroEraseDisc

All these methods first reset the error value and if an error occurred, the value is set accordingly.

```
NEROAPI_API void NADLL_ATTR NeroGetLastDriveError( NERO_DRIVE_ERROR
*driveError, void *reserved);
```

Identifier	Introduced in *NeroAPI* version
NeroGetLastDriveError	6.0.0.0

7.2.37. NeroGetLastError

If an error occurred, NeroGetLastError will return additional information.

```
NEROAPI_API char * NADLL_ATTR NeroGetLastError ();
```

Identifier	Introduced in *NeroAPI* version
NeroGetLastError	5.0.3.9

7.2.38. NeroGetLastErrors

This function returns recently logged errors. The number of returned errors is determined by iNum.

```
NEROAPI_API char * NADLL_ATTR NeroGetLastErrors(int iNum, DWORD dwFlags,
void *reserved);
```

Description of parameters		
iNum	Number of errors to be returned.	
dwFlags	available constants :	
	NGLE_ALL	(1<<0) Also include errors which do not contain a description to be shown in the GUI.
	NGLE_REPORT	(1<<1) Format the errors as in the *NeroAPI* error log.

Identifier	Introduced in *NeroAPI* version
NeroGetLastErrors	6.0.0.29

7.2.39. NeroGetLocalizedWaitCDTexts

Returned string must be released using NeroFreeMem. Function may return NULL if type is out of range.

Due to historical reasons, the function name refers to CD media. Please bear in mind that it can be used with DVD media as well!

```
NEROAPI_API char * NADLL_ATTR NeroGetLocalizedWaitCDTexts
(NERO_WAITCD_TYPE type);
```

Identifier	Introduced in *NeroAPI* version
NeroGetLocalizedWaitCDTexts	5.5.9.10

7.2.40. NeroGetTypeNameOfMedia

Get a string describing the given bit field of supported media. Free the string with NeroFreeMem.

```
NEROAPI_API char *NADLL_ATTR NeroGetTypeNameOfMedia(
                                                NERO_MEDIA_SET media,
                                                void *reserved);
```

Identifier	Introduced in *NeroAPI* version
NeroGetTypeNameOfMedia	5.0.3.9
NeroGetTypeNameOfMedia	5.5.9.4: Changed signature and behavior. Old version: NeroGetTypeNameOfMedia (DWORD media,const char *separator); With the current version, the separator of the current system language is used. This might cause problems if old code relies on the use of a special separator.

7.2.41. NeroGetVMSInfo

Virtual multisession is a technique to allow writing multisession discs on medium types that does not support normal multisession, e.g. DVD-/+RW.

This function retrieves virtual multisession information for media supporting it. NeroGetVMSInfo may be called for media having the NCDIMF_VIRTUALMULTISESSION flag set in their NERO_CD_INFO structure. Free the result with NeroFreeMem().

The function returns NULL in case of an error (e.g. non-VMS media inserted).

```
NEROAPI_API NERO_VMS_INFO * NADLL_ATTR NeroGetVMSInfo(
                                    NERO_DEVICEHANDLE aDeviceHandle,
                                    DWORD dwFlags);
```

Description of parameters	
aDeviceHandle	Device Handle, the result of NeroOpenDevice().
dwFlags	Currently unused, reserved for future extensions.

Identifier	Introduced in *NeroAPI* version
NeroGetVMSInfo	6.0.0.10

7.2.42. NeroGetWaitCDTexts

For a given NERO_WAITCD_TYPE a matching text message is returned.

This function is deprecated! Please use NeroGetLocalizedWaitCDTexts instead since it returns a localized string.

```
static const char *NeroGetWaitCDTexts (NERO_WAITCD_TYPE type)
```

7.2.43. NeroImportDataTrack

Create a NERO_ISO_ITEM tree from an already existing ISO track in order to create a new session with reference to files from older sessions.

```
NEROAPI_API NERO_ISO_ITEM *NADLL_ATTR NeroImportDataTrack(
                              NERO_DEVICEHANDLE pRecorder,
                              DWORD trackNumber,
                              void **ppCDStamp,
                              NERO_IMPORT_DATA_TRACK_INFO *pInfo,
                              DWORD flags,
                              NERO_IMPORT_DATA_TRACK_RESULT *result,
                              void *reserved);
```

Description of parameters		
pRecorder	First item of the root directory.	
trackNumber	Name of the CD.	
ppCDStamp	*ppCDStamp will be filled with a pointer on a CDStamp object which will have to be freed later.	
pInfo	Will be filled with information about the imported track.	
flags	Available constants :	
	NIITEF_IMPORT_ROCKRIDGE	(1<<0) Will be ignored, RockRidge is now always imported if present.
	NIITEF_IMPORT_ISO_ONLY	(1<<1)
	NIITEF_PREFER_ROCKRIDGE	(1<<2) Will be ignored.
	NIITEF_IMPORT_UDF	(1<<3) Import UDF Session.
	NIITEF_IMPORT_VMS_ SESSION	(1<<4) Treat trackNumber as the virtual multisession session specifier.
result	Will contain a result flag, may be NULL.	
reserved	Must be NULL.	

Identifier	Introduced in *NeroAPI* version
NeroImportDataTrack	5.9.9.9
pInfo	5.9.9.9
Result	5.9.9.9
NIITEF_IMPORT_VMS_SESSION	6.0.0.10

7.2.44. NeroImportIsoTrackEx

Create a NERO_ISO_ITEM tree from an already existing ISO track in order to create a new session with reference to files from older sessions.

This function is deprecated! Please Use NeroImportDataTrack instead!

```
NEROAPI_API NERO_ISO_ITEM *NeroImportIsoTrackEx(
                                        NERO_DEVICEHANDLE pRecorder,
                                        DWORD trackNumber,
                                        void **ppCDStamp,
                                        DWORD flags);
```

Description of parameters		
pRecorder	First item of the root directory.	
trackNumber	Name of the CD	
ppCDStamp	*ppCDStamp will be filled with a pointer on a CDStamp object which will have to be freed later	
flags	Available constants:	
	NIITEF_IMPORT_ROCKRIDGE	(1<<0) Will be ignored, RockRidge is now always imported if present.
	NIITEF_IMPORT_ISO_ONLY	(1<<1)
	NIITEF_PREFER_ROCKRIDGE	(1<<2) Will be ignored.
	NIITEF_IMPORT_UDF	(1<<3) Import UDF Session.
	NIITEF_IMPORT_VMS_ SESSION	(1<<4) Treat trackNumber as the virtual multisession session specifier.

Identifier	Introduced in *NeroAPI* version
NeroImportIsoTrackEx	5.0.3.9

7.2.45. NeroInit

Initialize the DLL. Must be successful before any of the remaining functions can be called. Settings structure and strings it points to are not copied and function callbacks must be available all the time. Provide NULL for the “reserved” parameter!

Make sure to keep **all** the data including the strings valid as long as you are using *NeroAPI*, since Nero will only store a pointer to the NERO_SETTINGS structure, not make a copy.

```
NEROAPI_API NEROAPI_INIT_ERROR NADLL_ATTR NeroInit (const NERO_SETTINGS
*pNeroSettings, const char *reserved);
```

Identifier	Introduced in *NeroAPI* version
NeroInit	5.0.3.9

7.2.46. NeroInitImageRecorder

Set the image file for the image recorder. This can be used to initialize the image recorder for packet writing.

If imageFilePath is NULL, the last opened file is closed.

```
NEROAPI_API int NADLL_ATTR NeroInitImageRecorder(
                              NERO_DEVICEHANDLE aDeviceHandle,
                              const char *imageFilePath,
                              DWORD dwFlags,
                              NERO_MEDIA_TYPE mediaType,
                              void *reserved);
```

Identifier	Introduced in *NeroAPI* version
NeroInitImageRecorder	6.0.0.25

7.2.47. NeroIsDeviceReady

This function returns a NERO_DRIVE_ERROR.

```
NEROAPI_API int NADLL_ATTR NeroIsDeviceReady(NERO_DEVICEHANDLE
aDeviceHandle);
```

Identifier	Introduced in *NeroAPI* version
NeroIsDeviceReady	5.0.3.9

7.2.48. NeroOpenDevice

Open a device. This function returns NULL if errors have occurred.

In general, an application may not access devices from multiple threads simultaneously. Even if NeroOpenDevice allows obtaining more than one handle for a device, the different handles may not be used at the same time.

However, here is an example of a case where it is legal to do so: When the *NeroAPI* calls the user dialog callback with DLG_NON_EMPTY_CDRW as type, it is permitted to delete the rewritable media with NeroEraseCDRW.

```
NEROAPI_API NERO_DEVICEHANDLE NADLL_ATTR NeroOpenDevice(const
NERO_SCSI_DEVICE_INFO* pDevInfo);
```

Identifier	Introduced in *NeroAPI* version
NeroOpenDevice	5.0.3.9

7.2.49. NeroRegisterDriveChangeCallback

Register a callback which is called whenever a drive was removed or added in the system. Use NeroGetAvailableDrivesEx to get the current list of drives in the system.

NOTE: In some rare cases the NeroAPI does not get this information from the OS and will therefore never notify the callback if a drive has been added or removed.

```
NEROAPI_API int NADLL_ATTR NeroRegisterDriveChangeCallback(
                                    NERO_DRIVESTATUS_CALLBACK callback,
                                    void *pUserData);
```

Description of parameters	
callback	The callback to be called when a drive is removed or added. The callback needs to be thread safe, since it might be called from a different thread.
pUserData	Data passed to the callback.

Identifier	Introduced in *NeroAPI* version
NeroRegisterDriveChangeCallback	6.0.0.0

7.2.50. NeroRegisterDriveStatusCallback

Register a callback which is called whenever the specified status of a drive is changed. This function returns 0 for success.

Please see the documentation of NERO_DRIVE_STATUS_TYPE for restrictions of the notifications.

```
NEROAPI_API int NADLL_ATTR NeroRegisterDriveStatusCallback(
                                NERO_DRIVESTATUS_TYPE status,
                                const NERO_SCSI_DEVICE_INFO *pDeviceInfo,
                                NERO_DRIVESTATUS_CALLBACK callback,
                                void *pUserData);
```

Description of parameters	
status	The status the application is interested in.
pDeviceInfo	The drive for which the status change should be notified. The pointer can be freed afterwards.
callback	The callback needs to be thread safe, since it might be called from a different thread.
pUserData	Data passed to the callback.

Identifier	Introduced in *NeroAPI* version
NeroRegisterDriveStatusCallback	6.0.0.0

7.2.51. NeroSetDesiredUDFRevision

Determine the desired UDF revision for a certain track.

```
NEROAPI_API BOOL NADLL_ATTR NeroSetDesiredUDFRevision(struct
CNeroIsoTrack *track, NERO_UDF_REVISION desiredRevision);
```

Description of parameters	
track	The track to perform the operation on.
desiredRevision	The desired revision as NERO_UDF_REVISION enum value.

Identifier	Introduced in *NeroAPI* version
NeroSetDesiredUDFRevision	6.6.0.13

7.2.52. NeroSetDesiredUDFPartitionType

Determine the desired UDF type for a certain track.

```
NEROAPI_API BOOL NADLL_ATTR NeroSetDesiredUDFPartitionType(struct
CNeroIsoTrack *track, NERO_UDF_PARTITION_TYPE desiredPartitionType);
```

Description of parameters	
track	The track to perform the operation on.
desiredPartition Type	The desired partion type as NERO_UDF_PARTITION_TYPE.

Identifier	Introduced in *NeroAPI* version
NeroSetDesiredUDFRevision	6.6.0.13

7.2.53. NeroSetDeviceOption

Set a special option for a device. Returns zero on success.

For example, by providing NERO_DEVICEOPTION_BOOKTYPE_DVDROM in the aOption parameter, the booktype can be changed to DVDROM - if the device allows it.

```
NEROAPI_API int NADLL_ATTR NeroSetDeviceOption(
                              NERO_DEVICEHANDLE aDeviceHandle,
                              NERO_DEVICEOPTION aOption,
                              void *value);
```

Description of parameters	
aDeviceHandle	The handle of a device.
aOption	A device option, e.g. setting the booktype.
Value	A pointer to an option specific type. E.g. when used to change the booktype, the parameter is expected to be BOOL*.

Identifier	Introduced in *NeroAPI* version
NeroSetDeviceOption	5.5.10.7

7.2.54. NeroSetExpectedAPIVersion

Using this function, an application can tell *NeroAPI* for which version of *NeroAPI* it was designed to work. *NeroAPI* then tries to behave like this version as much as possible. This ensures the binary compatibility with future versions of *NeroAPI*. If this function is not called, *NeroAPI* will behave as *NeroAPI 5.0.3.9*. If your application uses the NeroAPIGlue, this function will be called automatically.

Note: This function is obsolete since NeroAPI 5.5.9.9. Use NeroSetExpectedAPIVersionEx instead!

```
NEROAPI_API void NADLL_ATTR NeroSetExpectedAPIVersion(DWORD);
```

Identifier	Introduced in *NeroAPI* version
NeroSetExpectedAPIVersion	5.0.3.9

7.2.55. NeroSetExpectedAPIVersionEx

Using this function, an application can tell *NeroAPI* for which version of *NeroAPI* it was designed to work. *NeroAPI* then tries to behave like this version as much as possible. This ensures the binary compatibility with future versions of *NeroAPI*. If this function is not called, *NeroAPI* will behave as *NeroAPI 5.0.3.9*. If your application uses the NeroAPIGlue, this function will be called automatically.

It returns true for success. Provide NULL for the “reserved” parameter!

NeroSetExpectedAPIVersion was extended in *NeroAPI* 5.5.9.9 to support multiple digits.

If pPrevExpectedVersion is not NULL, it must point onto an array of 4 WORDs that will be filled with the previously expected version number.

```
NEROAPI_API BOOL NADLL_ATTR NeroSetExpectedAPIVersionEx( WORD majhi
                                                        ,WORD majlo
                                                        ,WORD minhi
                                                        ,WORD minlo
                                                        ,void reserved
                                                  ,WORD *pPrevExpectedVersion);
```

Identifier	Introduced in *NeroAPI* version
NeroSetExpectedAPIVersionEx	5.5.9.9
pPrevExpectedVersion	6.0.0.0

7.2.56. NeroSetOption

Used to set global Nero options such as the name of the message text file.

```
NEROAPI_API int NADLL_ATTR NeroSetOption(NEROAPI_OPTION option,void
*value);
```

Identifier	Introduced in *NeroAPI* version
NeroSetOption	5.0.3.9

7.2.57. NeroUpdateDeviceInfo

Update the information about a drive. The use of this function is only required when a drive was blocked by another application during identification (drive in use).

```
NEROAPI_API NERO_DRIVE_ERROR NADLL_ATTR NeroUpdateDeviceInfo(
                                        NERO_SCSI_DEVICE_INFO *devInfo,
                                        NERO_MEDIA_TYPE mediaType,
                                        void *reserved);
```

Description of parameters	
devInfo	The device info to update.
mediaType	The media type to update the speed information items with.
reserved	Must be NULL.

Identifier	Introduced in *NeroAPI* version
NeroUpdateDeviceInfo	6.0.0.0

7.2.58. NeroUnregisterDriveChangeCallback

Unregister a callback which was registered with NeroRegisterDriveChangeCallback.

```
NEROAPI_API int NADLL_ATTR NeroUnregisterDriveChangeCallback(
                                    NERO_DRIVESTATUS_CALLBACK callback,
                                    void *pUserData);
```

Description of parameters	
callback	The callback to be called when a drive is removed or added.
pUserData	Data passed to the callback.

Identifier	Introduced in *NeroAPI* version
NeroUnregisterDriveChangeCallback	6.0.0.0

7.2.59. NeroUnregisterDriveStatusCallback

Unregister a callback.

```
NEROAPI_API int NADLL_ATTR NeroUnregisterDriveStatusCallback(
                              NERO_DRIVESTATUS_TYPE status
                              const NERO_SCSI_DEVICE_INFO *pDeviceInfo,
                               NERO_DRIVESTATUS_CALLBACK callback,
                              void *pUserData);
```

Description of parameters	
status	The status the application is interested in.
pDeviceInfo	The drive for which the status was notified. The pointer does not need to be the same as in NeroRegisterDrivestatusCallback, but has to represent the same drive.
callback	The callback to be called if the status changed.
pUserData	Data passed to the callback.

Identifier	Introduced in *NeroAPI* version
NeroUnregisterDriveStatusCallback	6.0.0.0

7.2.60. NeroUserDlgInOut

This function gets a requester type and shall return a suitable response to it. Depending on the "type", "data" might contain additional information.

Argument passing is in standard C order (on the stack, right to left), also known as Microsoft Visual C++ __cdecl.

```
typedef NeroUserDlgInOut (NERO_CALLBACK_ATTR *NERO_USER_DIALOG)
                                                (void *pUserData,
                                                 NeroUserDlgInOut type,
                                                 void *data);
```

7.2.61. NeroWaitForDeviceReady

This function waits dwTimeOut milliseconds until a drive is ready. It returns a value of NERO_DRIVE_ERROR type.

```
NEROAPI_API int NADLL_ATTR NeroWaitForDeviceReady(NERO_DEVICEHANDLE
aDeviceHandle, DWORD dwTimeOut, void *reserved);
```

Description of parameters	
aDeviceHandle	Device handle.
dwTimeOut	timeout value.
reserved	Reserved.

Identifier	Introduced in *NeroAPI* version
NeroWaitForDeviceReady	6.6.0.2

7.2.62. NeroWaitForDisc

Use the nstUserDialog callback functions to request a media. Returns FALSE if the burn process should be aborted.

```
NEROAPI_API BOOL NADLL_ATTR NeroWaitForDisc(
                                    NERO_DEVICEHANDLE aDeviceHandle,
                                    NERO_MEDIA_SET nmt,
                                    DWORD dwBurnFlags,
                                    void *pCDStamp,
                                    DWORD dwFlags
                                    void *reserved);
```

Description of parameters	
nmt	Media types requested.
dwBurnFlags	Set of NBF_ flags.
pCDStamp	Optional stamp of requested media.
dwFlags	Set of NWFD_flags:
	(1<<0) NWFD_REQUIRE_EMPTY_DISC

Identifier	Introduced in *NeroAPI* version
NeroWaitForDisc	6.0.0.25

7.2.63. NeroWaitForMedia

Use the nstUserDialog callback functions to request a media. Returns FALSE if the burn process should be aborted.

This function is deprecated! Please use NeroWaitForDisc instead!

```
NEROAPI_API BOOL NADLL_ATTR NeroWaitForMedia (
                                    NERO_DEVICEHANDLE aDeviceHandle,
                                    NERO_MEDIA_SET nms,
                                    DWORD dwFlags,
                                    void *pCDStamp);
```

Description of parameters	
nms	Media types requested.
dwFlags	Set of NBF_ flags.
pCDStamp	Optional stamp of requested media.

Identifier	Introduced in *NeroAPI* version
NeroWaitForMedia	5.5.9.4

8. ISO Track Creation

When working with the *NeroAPI*, there are three ways for creating ISO tracks:

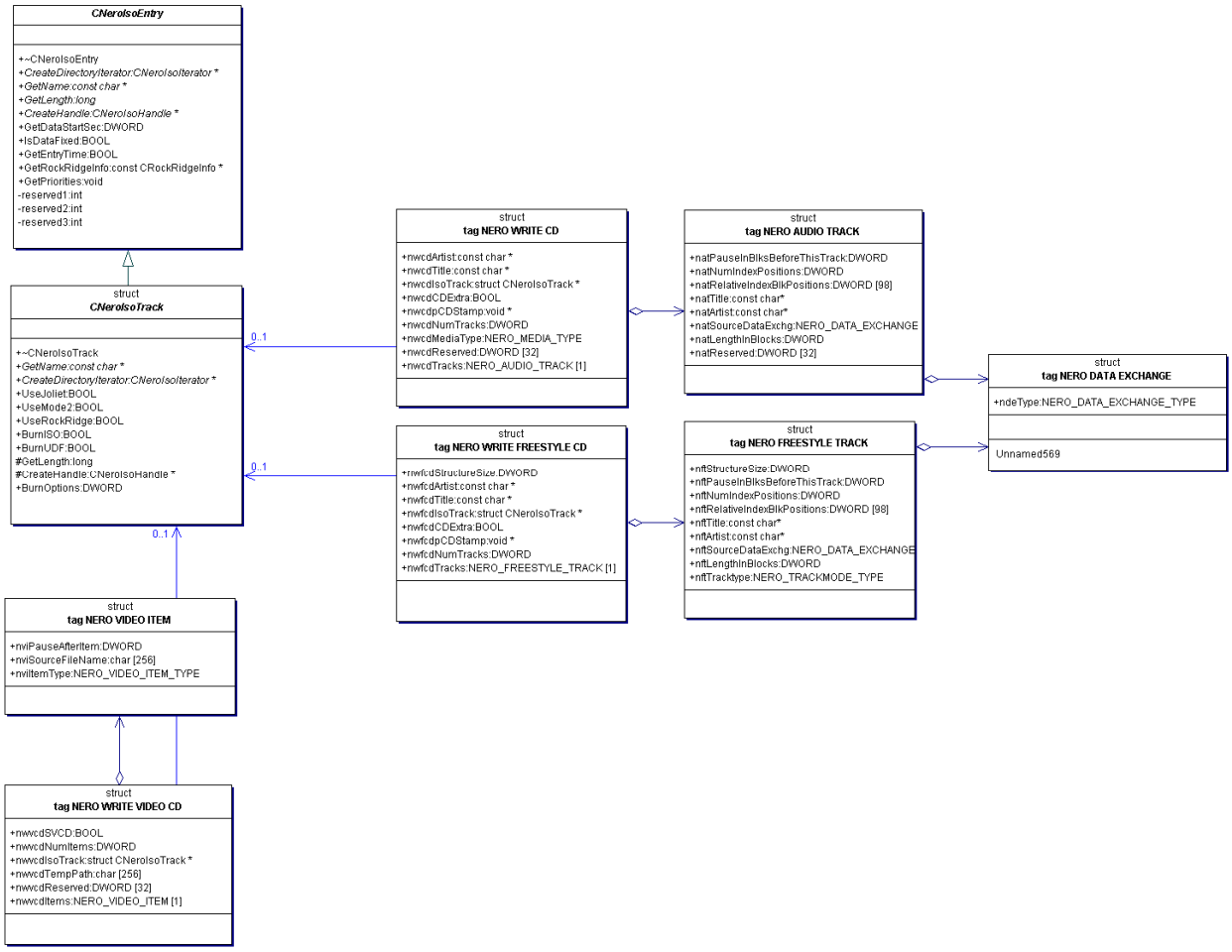
1. By creating classes derived from those declared in NeroIsoTrack.h, and putting a pointer to an instance into the NERO_WRITE_CD structure.
2. By creating a tree of NERO_ISO_ITEMs, and creating an CNeroIsoTrack object, using the NeroCreateIsoTrackEx function
3. By creating an instance of an IFileSystemDescContainer object, using the NeroCreateFileSystemContainer function, adding files to this object through the interfaces declared in NeroFileSystemContent.h, and then using the NERO_WRITE_FILE_SYSTEM_CONTENT structure.

These three interfaces have been created for different needs and coexist in *NeroAPI*.

9. ISO Track Classes

The following classes are used to write an ISO 9660/Joliet track. In contrast to most of the rest of the *NeroAPI*, the ISO Track interface is not written in pure C, but C++. Thus, the Nero ISO Track feature can only be used by C++ code.

9.1. Overview



9.2. CNeroDataCallback

The application has to specify the complete layout of the ISO track. The *NeroAPI* does not care at all where the data for the files comes from. This also means that the application has to provide access to the filename, or the data itself, when the API needs it. Data can be fed into the API directly (i.e. without intermediate files) with CNeroDataCallback.

```
class CNeroDataCallback
{
public:
   virtual ~CNeroDataCallback () {}
   virtual DWORD IOCallback(BYTE *pBuffer, DWORD dwLen) = 0;
   virtual BOOL EOFCallback ()                          = 0;
   virtual BOOL ErrorCallback ()                        = 0;
};
```

Description of class member functions	
IOCallback	same semantic as NERO_IO_CALLBACK in "NeroAPI.h"
EOFCallback	same semantic as NERO_IO.nioEOFCallback
ErrorCallback	same semantic as NERO_IO.nioErrorCallback

9.3. CNeroIsoHandle

The API builds an internal representation of the complete ISO tree and uses a CNeroIsoHandle, acquired from the application for each file, to access the data later.

```
class CNeroIsoHandle
{
public:
   virtual ~CNeroIsoHandle () {}
   virtual CNeroIsoHandle * Clone ()                                = 0;
   virtual int GetFileName (char *strBuffer, UINT nBufferSize)      = 0;
   virtual CNeroDataCallback * Open ()                              = 0;
};
```

Description of class member functions	
Clone	Creates a copy of the CNeroIsoHandle object.
GetFileName	If the application wants the API to read files, it has to fill the buffer of size nBufferSize with a null-terminated string and return the length of the full name, even if the given buffer was too small. The API will try again with a larger buffer then. Return 0 in GetFileName if you want to provide the data via a CNeroDataCallback
Open	Return instance ready to read the data associated with this handle or NULL for error; this instance will be deleted by the *NeroAPI*; usually only one file at once will be left open

9.4. CNeroIsoIterator

Iterators are used to walk through directories while the API builds its internal copy of the tree. Iterators point to an entry or to NULL, if the last entry was passed, and can only be incremented.

```
class CNeroIsoEntry;
class CNeroIsoIterator
{
public:
   virtual ~CNeroIsoIterator () {}
   virtual CNeroIsoEntry * GetCurrentEntry ()     = 0;
   virtual void Next ()                           = 0;
};
```

Description of class member functions	
GetCurrentEntry	Get pointer to current entry or NULL if last one passed; entry not deleted by API, so the iterator may point to itself and implement the required interface (as in the *NeroAPI* demo), or to some permanent entry.
Next	Go to next entry.

9.5. CNeroIsoEntry

```
struct CImportInfo;
class CNeroIsoEntry
{
public:
   virtual ~CNeroIsoEntry () {}

   virtual CNeroIsoIterator * CreateDirectoryIterator() = 0;
   virtual const char *       GetName ()                = 0;
   virtual __int64            GetLength ()              = 0;
   virtual CNeroIsoHandle *   CreateHandle ()           = 0;
   virtual DWORD GetDataStartSec() { return 0;}
   virtual BOOL IsDataFixed() { return FALSE;}
   virtual BOOL GetEntryTime(struct tm *tm) {return FALSE;}
   virtual const CImportInfo *GetImportInfo() const
    {return NULL;};
   virtual void GetPriorities(int &iPriority,int &iDirPriority)
   {
     iPriority      =0;
     iDirPriority   =0;
   };
   virtual CNeroIsoIterator * CreateDirectoryIteratorWrapper()
                                { return NULL;    }
   virtual CNeroIsoHandle *   CreateResourceHandle ()
                                { return NULL; };
   virtual const WCHAR* GetUnicodeName() { return 0; } //
```

```
private:

   virtual NERO_UDF_REVISION       GetDesiredUDFRevision()
                                            { return NUR_UDF_INVALID; }
   virtual NERO_UDF_PARTITION_TYPE GetDesiredUDFPartitionType()
                                            { return NUPT_INVALID;    }
   virtual int reserved1()      {return 0;}
};
```

Description of class member functions	
CreateDirectoryIterator	NULL if no directory, otherwise an iterator to step through all child entries; iterator will be deleted by the *NeroAPI*.
GetName	The name for this entry; will be copied by API.
GetLength	The size of this entry in bytes, or -1 if a directory.
CreateHandle	Creates a handle stored by the API to open a file later, NULL for directory; handle will be deleted by *NeroAPI* when deleting the internal ISO tree.
GetDataStartSec	Can be used to reference files from previous session. Not fully implemented yet.
IsDataFixed	Can be used to reference files from previous session.
GetEntryTime	Can be used to reference files from previous session.
GetImportInfo	This method was formerly known as GetRockRidgeInfo. The object returned is a bit different internally now. Since it is a private structure of *NeroAPI* this change does not matter. No ImportInfo by default
GetPriorities	
CreateDirectoryIterator Wrapper	This function is equivalent to CreateDirectoryIterator but returns an iterator for the wrapper file system of a CD, e.g. when creating HFS+ CDs with an HFS wrapper file system.
CreateResourceHandle	See CreateHandle. Creates rsc fork handle for HFS file systems. Will be preferred to reading the resource fork from the file specified by GetName if different from NULL.
GetUnicodeName	The name for this entry in Unicode format; will be copied by the API.
GetDesiredUDFRevision	Returns the desired UDF revision.
GetDesiredUDFPartition Type	Returns the desired UDF partition type.
reserved1	Reserved for future use.

Identifier	Introduced in *NeroAPI* version
CreateDirectoryIteratorWrapper	5.5.9.0
CreateHandle	5.5.9.0
GetUnicodeName.	6.0.0.0
GetDesiredUDFRevision	6.6.0.13
GetDesiredUDFPartitionType	6.6.0.13

9.6. CNeroIsoTrack

An ISO track is a special directory entry.

```
struct CNeroIsoTrack : public CNeroIsoEntry
{
   friend class CNeroIsoTrackProxy5039;
   friend class CNeroIsoTrackProxy55915;

public:
   ~CNeroIsoTrack () {}

   virtual const char *       GetName () = 0;
   virtual CNeroIsoIterator * CreateDirectoryIterator () = 0;

   virtual BOOL               UseJoliet () { return TRUE; }
   virtual BOOL               UseMode2 () { return FALSE; }

   virtual BOOL               UseRockRidge () { return FALSE; }
   virtual BOOL               BurnISO() { return TRUE; }
   virtual BOOL               BurnUDF() { return FALSE; }

protected:
   virtual __int64            GetLength () { return -1; }
   virtual CNeroIsoHandle *   CreateHandle () { return NULL; }
   virtual CNeroIsoHandle *   CreateResourceHandle () { return NULL; }

public:
   virtual DWORD              BurnOptions()
     {
         return (UseJoliet() ? NCITEF_USE_JOLIET : 0)
                 |(UseMode2() ? NCITEF_USE_MODE2 : 0)
                 |(UseRockRidge() ? NCITEF_USE_ROCKRIDGE : 0)
                 |(BurnISO() ? NCITEF_CREATE_ISO_FS : 0)
                 |(BurnUDF() ? NCITEF_CREATE_UDF_FS : 0);
     };

   virtual CNeroIsoIterator *CreateDirectoryIteratorWrapper()
   { return NULL; }

   virtual BOOL HasWrapper(void)
   { return FALSE; }

   virtual const void *dummy() const { return NULL; };
```

```
   virtual void GetVolumeDescriptor(const char **systemIdentifier,
                    const   char **volumeSet, const char **publisher,
                    const char **dataPreparer, const char **application,
                    const char **copyright, const char **abstract,
                    const char **bibliographic)
   {
         *systemIdentifier = 0;
         *volumeSet = 0;
         *publisher = 0;
         *dataPreparer = 0;
         *application = 0;
         *copyright = 0;
         *abstract = 0;
         *bibliographic = 0;
   }

   virtual NERO_UDF_REVISION       GetDesiredUDFRevision()
                                            { return NUR_UDF_INVALID; }
   virtual NERO_UDF_PARTITION_TYPE GetDesiredUDFPartitionType()
                                            { return NUPT_INVALID;    }

   virtual int reserved1() { return 0;}
   virtual int reserved2() { return 0;}
   virtual int reserved3() { return 0;}
   virtual int reserved4() { return 0;}
   virtual int reserved5() { return 0;}
   virtual int reserved6() { return 0;}
   virtual int reserved7() { return 0;}
   virtual int reserved8() { return 0;}
};
```

Description of class member functions	
GetName	Essential function. ISO volume name, copied by API.
CreateDirectoryIterator	Essential function. Iterator for root directory; will be deleted by API.
UseJoliet	Function returns reasonable default. TRUE if track shall contain Joliet names in addition to ISO.
UseMode2	Function returns reasonable default. TRUE if track shall be written as mode 2/XA.
UseRockRidge	Function returns reasonable default. RockRidge requires additional information, so it is off by default.
BurnISO	Function exists from *NeroAPI* version 5.5.0.0. TRUE if ISO should be created.
BurnUDF	Function exists from *NeroAPI* version 5.5.0.0. TRUE if UDF should be created.
GetLength	Function exists from *NeroAPI* version 5.5.0.0. Returns value for directory.
CreateHandle	Return NULL, so object cannot be read.
CreateResourceHandle	CNeroIsoTrack is a special directory, so no file handle is available and NULL is returned.
BurnOptions	From *NeroAPI* version 5.5.1.2 you can set your burn options simply by redefining this function instead of UseJoliet, UseMode2, UseRockRidge, BurnISO and BurnUDF. See 7.2.12 NeroCreateIsoTrackEx for the significance of the NCITEF flags.
dummy	This method is for internal use only. Do not reimplement it!
GetVolumeDescriptor	The *NeroAPI* will call this method to determine whether that information exists. If it does, it will be used during the creation of the media. To provide this information, either derive a class from CNeroIsoTrack and overwrite this function or provide it when you call NeroCreateIsoTrack. This information is identical to what Nero displays on the “Label” tab of an ISO compilation.
GetDesiredUDFRevision	Returns the desired UDF revision.
GetDesiredUDFPartition Type	Returns the desired UDF partition type.

Identifier	Introduced in *NeroAPI* version
CreateDirectoryIteratorWrapper	5.5.9.0
HasWrapper	5.5.9.0
CreateResourceHandle	5.5.9.2
GetBootInfo	5.5.9.16
GetVolumeDescriptor	5.5.10.2
GetDesiredUDFRevision	6.6.0.13
GetDesiredUDFPartitionType	6.6.0.13

10. The FileSystemContent Interface

This is the third *NeroAPI* interface for preparing data CDs/DVDs. Unlike NeroIsoTrack.h, it is not much "callback based", thus most of the process will be driven by the application, making it easier to write. This interface is closely connected to the internal engine of *NeroAPI*; this improves the cooperation of *NeroAPI* and the application.

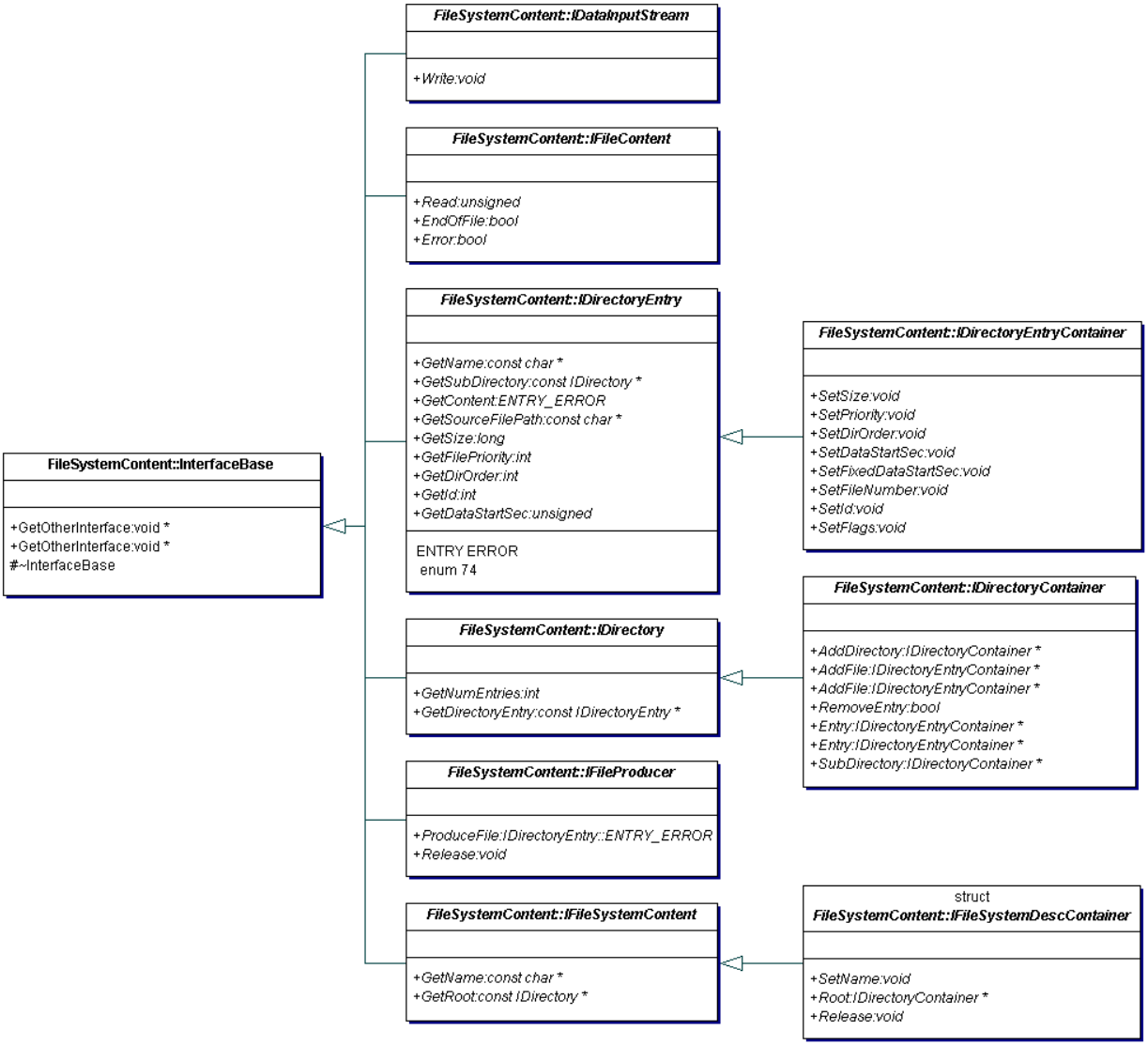
This set of classes describes the content of the file system of a disc. The application will build a file structure using the IFileSystemContent object.

During the burn process, *NeroAPI* will request the content of files using the IFileContent interface.

Use the NeroCreateFileSystemContainer function of NeroAPI.h to get an instance of an IFileSystemDescContainer object.

Then, use the NERO_WRITE_FILE_SYSTEM_CONTAINER structure to burn the file structure created.

10.1. Overview



10.2. Namespace setting

To make sure we do not interfere with other classes we use the namespace “FileSystemContent” for this group of interfaces.

```
namespace FileSystemContent
```

10.3. InterfaceBase

GetOtherInterface returns a different interface for the same object. This will be used to extend the DLL interface without loosing binary compatibility. The function returns NULL if no interface with this ID was found. This is inspired from the COM QueryInterface function.
A different interface can be requested by providing an ID number or a string. Currently no other interfaces are available by default.

```
class InterfaceBase
{
public:
  virtual void *GetOtherInterface(int interfaceId)            const
  {
          return 0;
  }
  virtual void *GetOtherInterface(const char *interfaceName)  const
  {
          return 0;
  }

protected:
  virtual ~InterfaceBase() {}
};
```

10.4. File System Reading Interfaces

This first set of interfaces will be used by the burn engine to read the content of the file system.

10.4.1. IFileContent

Release will be called by the application when the object is not needed anymore

```
class IFileContent : public InterfaceBase
{
public:
      virtual unsigned Read(void *pBuffer,unsigned length) =0;
      virtual bool EndOfFile()                             =0;
      virtual bool Error()                                 =0;

      virtual void Release()                               =0;
};
```

10.4.2. IDirectoryEntry

GetName returns a file or directory name.

GetSourceFilePath returns the source file path, NULL if the file is generated.

GetId returns an Id number that can be used to find the file again later.

```
class IDirectoryEntry : public InterfaceBase
{
public:
      enum ENTRY_ERROR
      {
            ENTRY_NO_ERROR,
            SEQUENCING_ERROR,
            ERROR_NOT_A_FILE,
            NOT_AVAILABLE,
            INTERFACE_ERROR
      };

      enum
      {
            MODE2_FORM2                   =1<<0,
            FIXED_INSIDE_VOLUME_SPACE     =1<<1,
            FIXED_OUTSIDE_VOLUME_SPACE    =1<<2,
            NO_OWN_CONTENT                =1<<3,
            CPR_MAI_ADPTY                 =1<<4
      };

      virtual const char *GetName()                   const =0;
```

```
      virtual const IDirectory *GetSubDirectory()    const =0;
      virtual ENTRY_ERROR GetContent(IFileContent **)const =0;
      virtual const char *GetSourceFilePath()        const =0;
      virtual __int64   GetSize()                    const =0;
      virtual int GetFilePriority()                  const =0;
      virtual int GetDirOrder()                      const =0;
      virtual int GetId()                            const =0;
      virtual unsigned GetDataStartSec()             const =0;
};
```

Description of enumerators	
SEQUENCING_ERROR	Indicates that the content for this file may not be requested at this moment.
ERROR_NOT_A_FILE	This entry is not a file
NOT_AVAILABLE	The content of this file cannot be requested at all.
INTERFACE_ERROR	The overridden function has tried to get another interface for one object and has failed.
FEATURE_NOT_AVAILABLE	This feature is not available for this file system type.

Identifier	Introduced in *NeroAPI* version
FEATURE_NOT_AVAILABLE	5.5.8.2
NO_OWN_CONTENT	5.5.9.4

10.4.3. IDirectory

```
class IDirectory : public InterfaceBase
{
public:
      virtual int GetNumEntries()                            const =0;
      virtual const IDirectoryEntry *GetDirectoryEntry(int i) const =0;
};
```

10.4.4. IFileSystemContent

GetName returns the volume name.

```
class IFileSystemContent : public InterfaceBase
{
public:
      virtual const char *GetName()      const =0;
      virtual const IDirectory *GetRoot()const =0;
};
```

10.5. File System Content Creation Interfaces

This second set of interfaces will be used by the application to produce the content of the file system.

10.5.1. IDataInputStream

This interface allows the file producer to return the data.

```
class IDataInputStream : public InterfaceBase
{
public:
      virtual void Write(const void *buffer,int size)= 0;
};
```

10.5.2. IFileProducer

Produce the content of a file. This interface must be derived and its implementation must create the content of the file in the ProduceFile function.

Calling ProduceFile will automatically update the file size to the amount of data delivered by the producer.

Release will be called by the *NeroAPI* when the object is not needed anymore

```
class IFileProducer : public InterfaceBase
{
public:
      virtual IDirectoryEntry::ENTRY_ERROR ProduceFile(
              IDataInputStream *str) const = 0;

      virtual void Release()        const = 0;
};
```

10.5.3. IDirectoryEntryContainer

This interface provides the means of describing a file. Using the SetSize function, the file size can be changed after having added the entry to the directory.

If the file entry was created using an IFileProducer object, this one can be retrieved using GetOtherInterface.

SetDataStartSec sets the sector number that will be saved into the directory structure.

SetFixedDataStartSec sets the physical position of the file in the file system.

SetPriority and SetDirOrder can be used to re-adjust the directory priority. Priority numbers will be used in upward order: the file with smaller values first.

SetFlags enables or disables the given flag.

```
class IDirectoryEntryContainer : public IDirectoryEntry
{
public:
      enum
      {
        IID_IDirectoryEntryContainer,
        IID_IFileProducer,
        IID_IDirectoryEntryContainer2,
        IID_IDirectoryEntry2
      };
      virtual void SetSize(__int64 size)                  =0;

      virtual void SetPriority(int priority)              =0;
      virtual void SetDirOrder(int directoryPriority)     =0;
      virtual void SetDataStartSec(unsigned)              =0;
      virtual void SetFixedDataStartSec(unsigned)         =0;
      virtual void SetFileNumber(int)                     =0;
      virtual void SetId(int)                             =0;
      virtual void SetFlags(bool enable,unsigned f)       =0;
};
```

10.5.4. IDirectoryContainer

This interface represents the content of a directory. AddDirectory returns a pointer to the directory; directoryPriority specifies the position in the directory. AddFile adds a file to the directory. The fp object will be automatically deleted when the directory container will be deleted.

The file size passed here does **not** need to be correct; it will be used by the file system generator to pre-allocate space so it must be the **maximum** space the final version of the file may need (worst-case).

Priority specifies some user-defined ordinal defining the order in which the files are being written to the disc physically (for example, .ifo comes before .vob).

Priorities are valid across directories. The fileentry order in a directory is defined by the directoryPriority parameter which is the primary sort criterion when arranging the files in a directory (Note that this is only true for file systems that do not require files to be sorted in the directory, e.g. UDF).

If any of the priority specifiers is -1, the producer does not care about the priority and the *NeroAPI* will put the file where it thinks it fits. AddFile will add a file which is present in the real file system, and return NULL if a file with the same name already exists.

RemoveEntry removes an entry from the directory.

```
class IDirectoryContainer : public IDirectory
{
public:
      virtual IDirectoryContainer *AddDirectory(const char *name,
              int directoryPriority)                              =0;

      virtual IDirectoryEntryContainer *AddFile(const char *name,
              const IFileProducer *fp,__int64 size,int priority,
              int directoryPriority)                              =0;


      virtual IDirectoryEntryContainer *AddFile(const char *name,
              const char *sourcePath, int priority,
              int directoryPriority)                              =0;

      virtual bool RemoveEntry(const char *name)                  =0;

      virtual IDirectoryEntryContainer *Entry(const char *name)  =0;
      virtual IDirectoryEntryContainer *Entry(int i)             =0;
      virtual IDirectoryContainer *SubDirectory(const char *name =0;
};
```

10.5.5. IFileSystemDescContainer

This interface represents the content of a file system.

SetName sets the volume name of the file system.

The Root function provides access to the root directory for changing it.

Release is called by the application when the object is not needed anymore.

```
struct IFileSystemDescContainer : public IFileSystemContent
{
      virtual void SetName(const char *) =0;
      virtual IDirectoryContainer *Root()=0;

      virtual void Release()       const =0;
};
```

10.5.6. IFileSystemDescContainerVolume

This is an extension of IFileSystemDescContainer to set extended volume names.

If you use these methods, do not use SetName of IFileSystemDescContainer.

```
struct IFileSystemDescContainerVolume : public InterfaceBase
{
      virtual void SetISOName(const char *) = 0;
      virtual void SetJolietName(const WCHAR *) = 0;
      virtual void SetUDFName(const WCHAR *) = 0;
```

Non-const get functions will return a volume label even if it was not set by calling above methods. These methods will provide a volume label that will actually allow burning the disc by converting illegal characters.

```
      virtual const char*  GetISOName() = 0;
      virtual const WCHAR* GetJolietName() = 0;
      virtual const WCHAR* GetUDFName() = 0;
};
```

11. The Burn-at-once Interface

The Burn-at-once interface allows burning a DVD-Video without storing intermediate files on your hard disk.

There are two ways to use this functionality:

1. Calling the NeroBurnAtOnce function and responding to callbacks from the *NeroAPI.*
2. Calling the NeroBAOxxx methods. Here the application controls the process. The downside is that the application has to ensure that data bits are provided in the right order while burning the DVD.

Due to the way the Burn-at-once process writes the data to a DVD, there might be small gaps between some files on the disc. Therefore, the navigation data items in the IFO files need to be adapted to the actual locations of the files on the disc. The *NeroAPI* will provide you with the necessary information during the Burn-at-once burn process.

11.1. The NERO_WRITE_BURN_AT_ONCE struct

This structure is passed to the NeroBurnAtOnce and NeroBAOCreateHandle functions as parameter.

```
typedef struct tag_NERO_BURN_AT_ONCE
{
   DWORD nwbaoSize;
   const char *nwbaoTempDirectory;
#ifdef __cplusplus
FileSystemContent::
#else //__cplusplus
struct
#endif//__cplusplus
   IFileSystemDescContainer *nwbaoFSContainer;
   DWORD nwbaoReserved[64];
} NERO_WRITE_BURN_AT_ONCE;
```

Description of structure members	
nwbaoSize	Fill this with sizeof(NERO_BURN_AT_ONCE).
nwbaoTempDirectory	Path to a directory to store temporary files. If NULL, the system temp directory will be used.
nwbaoFSContainer	The content of the file system. **Important:** All IFO and BUP files must have the exact file size set. All other video files must have the estimated size set. The size will be updated during burning. Note, that the estimated size should be as close to the actual size as possible.

Description of structure members	
	For each video title set only add the first VOB file (e.g. VTS_01_1.VOB) to the file system container and pass all the video data of this title set in the file producer of this file. The *NeroAPI* will automatically split the file at the appropriate position (1 GB - logical block size). DVD-Video files will be sorted automatically. It is guaranteed that the file data will be requested in the following order: • VMGM_VOB (VIDEO_TS.VOB) [if present] • For each video title set (VTS): · VTSM_VOB (e.g. VTS_01_0.VOB) [if present] · VTSTT_VOBS (e.g. VTS_01_1.VOB) [mandatory] · VTSI file (e.g. VTS_01_0.IFO) [mandatory] · VTSI backup (e.g. VTS_01_0.BUP) [mandatory] • VMGMI (VIDEO_TS.IFO) [mandatory] • Backup for VMGMI (VIDEO_TS.BUP) [mandatory]
nwbaoReserved	Should be zero.

Identifier	Introduced in *NeroAPI* version
NERO_WRITE_BURN_AT_ONCE	6.3.1.11

11.2. The IBurnAtOnceInfo Interface

Access this interface through the IDataInputStream object you get in IFileProducer::ProduceFile. Use GetOtherInterface("IBurnAtOnceInfo") to get a pointer to an object of this class.

```
class IBurnAtOnceInfo : public InterfaceBase
{
public:
   virtual DWORD GetOffset() const = 0;
};
```

11.2.1. GetOffset

The GetOffset method returns the offset from the start of VIDEO_TS.IFO in blocks.

11.3. Functions

11.3.1. NeroBurnAtOnce

Start the Burn-at-once process. The *NeroAPI* will use callbacks to request data.

```
NEROAPI_API NEROAPI_BURN_ERROR NADLL_ATTR NeroBurnAtOnce(
                        NERO_DEVICEHANDLE        aDeviceHandle,
                        NERO_WRITE_BURN_AT_ONCE* pBurnAtOnce,
                        DWORD                    dwFlags,
                        DWORD                    dwSpeed,
                        NERO_PROGRESS*           pNeroProgress,
                        void*                    reserved);
```

Description of parameters	
aDeviceHandle	Device handle.
pBurnAtOnce	Pointer to a NERO_WRITE_BURN_AT_ONCE struct.
dwFlags	NBF flags. Note that not all NBF flags will be interpreted.
dwSpeed	In KB/s if NBF_SPEED_IN_KBS is present, in multiple of 150 KB/s otherwise.
pNeroProgress	Nero progress callback.
reserved	Set this to NULL.

Identifier	Introduced in *NeroAPI* version
NeroBurnAtOnce	6.3.1.11

11.3.2. NeroBAOCreateHandle

This function is part of the second method to do Burn-at-once with the *NeroAPI*.

It allows more control by the application but the caller has to make sure that the data is provided in the correct order. Additionally, the *NeroAPI* will not cache any data in memory with this method and the functions block until all the data is written. Therefore, your application should implement some caching itself for performance reasons.

Use it like this:

1. Fill the nwbaoFSContainer of NERO_WRITE_BURN_AT_ONCE to specify the layout of the DVD.
2. Call NeroBAOCreateHandle with the necessary information to get a NERO_BAO_HANDLE.
3. For each video file in the VIDEO_TS folder, call NeroBAOOpenFile, then write all the data with NeroBAOWriteToFile and then call NeroBAOCloseFile.

Note: It is absolutely important, that this is done in the correct order. See the remarks on the nwbaoFSContainer above.

4. After all video data is written, call NeroBAOCloseHandle. This will write additional files that are present in the file system container and then finalize the disc.
5. If an error occurs on your side and/or you want to cancel burning, just call NeroBAOCloseHandle with dwFlags set to NBAOF_CANCELED or NBAOF_FAILED.
6. If an error occurs on the *NeroAPI* side, also call NeroBAOCloseHandle to end burning and do some cleanup.

Notes: nwbaoFSContainer of parameter pBurnAtOnce is required to be valid until NeroBAOCloseHandle is called.

```
NEROAPI_API NEROAPI_BURN_ERROR NADLL_ATTR NeroBAOCreateHandle(
                        NERO_DEVICEHANDLE        aDeviceHandle,
                        NERO_WRITE_BURN_AT_ONCE* pBurnAtOnce,
                        DWORD                    dwFlags,
                        DWORD                    dwSpeed,
                        NERO_PROGRESS*           pNeroProgress,
                        NERO_BAO_HANDLE*         pBAOHandle,
                        void*                    reserved);
```

Description of parameters	
aDeviceHandle	Device handle.
pBurnAtOnce	Pointer to a NERO_WRITE_BURN_AT_ONCE struct.
dwFlags	NBF flags. Note that not all NBF flags will be interpreted.
dwSpeed	In KB/s if NBF_SPEED_IN_KBS is present, in multiple of 150 KB/s otherwise.
pNeroProgress	Nero progress callback.
pBAOHandle	Will receive the handle created by *NeroAPI*.

Identifier	**Introduced in *NeroAPI* version**
NeroBAOCreateHandle	6.3.1.11

11.3.3. NeroBAOOpenFile

```
NEROAPI_API NEROAPI_BURN_ERROR NADLL_ATTR NeroBAOOpenFile(
                                    NERO_BAO_HANDLE   hBAOHandle,
                                    const char*       name,
                                    DWORD*            pdwOffset,
                                    void*             reserved);
```

Description of parameters	
hBAOHandle	The handle that the *NeroAPI* created as result of NeroBAOCreateHandle.
name	File name.
pdwOffset	Contains the offset of the file from the beginning of VIDEO_TS.IFO in blocks. (This is equivalent to IBurnAtOnceInfo::GetOffset when using the other method.)
reserved	Set this to NULL.

Identifier	Introduced in *NeroAPI* version
NeroBAOOpenFile	6.3.1.11

11.3.4. NeroBAOWriteToFile

```
NEROAPI_API NEROAPI_BURN_ERROR NADLL_ATTR NeroBAOWriteToFile(
                                NERO_BAO_HANDLE   hBAOHandle,
                                const void*       lpBuffer,
                                DWORD             nNumberOfBytesToWrite
                                LPDWORD           lpNumberOfBytesWritten,
                                void*             reserved);
```

Description of parameters	
hBAOHandle	The handle, which the *NeroAPI* created as result of NeroBAOCreateHandle.
lpBuffer	Data buffer.
nNumberOfBytes ToWrite	Number of bytes to write. Must be a multiple of 2048!
lpNumberOf BytesWritten	Number of bytes actually written.
reserved	Set this to NULL.

Identifier	Introduced in *NeroAPI* version
NeroBAOWriteToFile	6.3.1.11

11.3.5. NeroBAOCloseFile

```
NEROAPI_API NEROAPI_BURN_ERROR NADLL_ATTR NeroBAOCloseFile(
                                          NERO_BAO_HANDLE   hBAOHandle,
                                          void*             reserved);
```

Description of parameters	
hBAOHandle	The handle, which the *NeroAPI* created as result of NeroBAOCreateHandle.
reserved	Set this to NULL.

Identifier	Introduced in *NeroAPI* version
NeroBAOCloseFile	6.3.1.11

11.3.6. NeroBAOCloseHandle

Use the dwFlags parameter to instruct the *NeroAPI* to terminate the process. The reason might be the user's desire to cancel the operation or a failure. If one of the flags is present, *NeroAPI* will not attempt to write additional data but stop the burn process immediately.

```
NEROAPI_API NEROAPI_BURN_ERROR NADLL_ATTR NeroBAOCloseHandle(
                                          NERO_BAO_HANDLE   hBAOHandle,
                                          DWORD             dwFlags,
                                          void*             reserved);
```

Description of parameters		
hBAOHandle	The handle, which the *NeroAPI* created as result of NeroBAOCreateHandle.	
dwFlags	Available constants:	
	NBAOF_CANCELED	(1<<0)
	NBAOF_FAILED	(1<<1)
reserved	Set this to NULL.	

Identifier	Introduced in *NeroAPI* version
NeroBAOCloseHandle	6.3.1.11

12. The Packet Writing API

Packet writing enables the incremental writing of data to a CD-R or DVD. Unlike disk-at-once or track-at-once it lets the user access the media like a hard disk drive if the CD or DVD recorder supports packet writing.

Packet writing has become available with *NeroAPI* 5.5.10.15.

12.1. Packet Writing Interface

Please note that only one object created by either NeroCreateBlockWriterInterface or NeroCreateBlockReaderInterface may exist at a time. Also make sure to delete the object before using the referred drive for another purpose (e.g. importing multisession data, starting a recording- or digital audio extraction process). Opening a secondary device handle for the drive is **not** sufficient!

12.1.1. Access Mode

The enum is used when creating reader or writer interfaces from devices.

```
typedef enum
{
      eNoWriting              =0x0000,
      ePacketWriting          =0x0001,
      eManagedMRW             =0x0002,
      eRawMRW                 =0x0004,
      eIllegalAccessMode      =0xffffffff
} AccessMode;
```

Description of enumerators	
eNoWriting	Use this to instantiate an INeroFileSystemBlockAccess object for read-only access.
ePacketWriting	Use this for DVD+RW, DVD-RW, CD-RW media in non-MRW mode.
eManagedMRW	Use this for defective managed MRW mode for all media types.
eRawMRW	Use this for raw MRW mode (defective management turned off).

12.1.2. ImageAccessMode

Used when creating a block access interface from an image.

```
typedef enum
{
      eIAReadOnly             = 0x0000,
      eIAReadWrite            = 0x0001,
      eIAIllegalAccessMode    = 0xffffffff
} ImageAccessMode;
```

Description of enumerators	
eIAReadOnly	Read only access.
eIAReadWrite	Read and write access.

12.1.3. NeroCreateBlockWriterInterface

Use this function to obtain a block writer to a specified *NeroAPI* device.

Please note that ownership of the aDeviceHandle is **not** transferred to the block writer so you are still responsible to dispose of the device handle after disposing of the writer interface.

```
NEROAPI_API INeroFileSystemBlockAccess* NADLL_ATTR
NeroCreateBlockWriterInterface( NERO_DEVICEHANDLE DeviceHandle,
                                AccessMode eAccessMode);
```

12.1.4. NeroCreateBlockReaderInterface

Use this function to obtain a block reader to a specified *NeroAPI* device.

Please note that ownership of the aDeviceHandle is **not** transferred to the block reader so you are still responsible to dispose of the device handle after disposing of the reader interface.

```
NEROAPI_API INeroFileSystemBlockReader* NADLL_ATTR
NeroCreateBlockReaderInterface( NERO_DEVICEHANDLE DeviceHandle,
                                AccessMode eAccessMode);
```

12.1.5. NeroCreateBlockAccessFromImage

Create a block access interface for the specified image file. Instead of an image file, you may pass a drive letter here to read from a specific device supported by the operating system.

```
NEROAPI_API INeroFileSystemBlockAccess* NADLL_ATTR
NeroCreateBlockAccessFromImage(const char *szFilename,
                               ImageAccessMode eAccessMode);
```

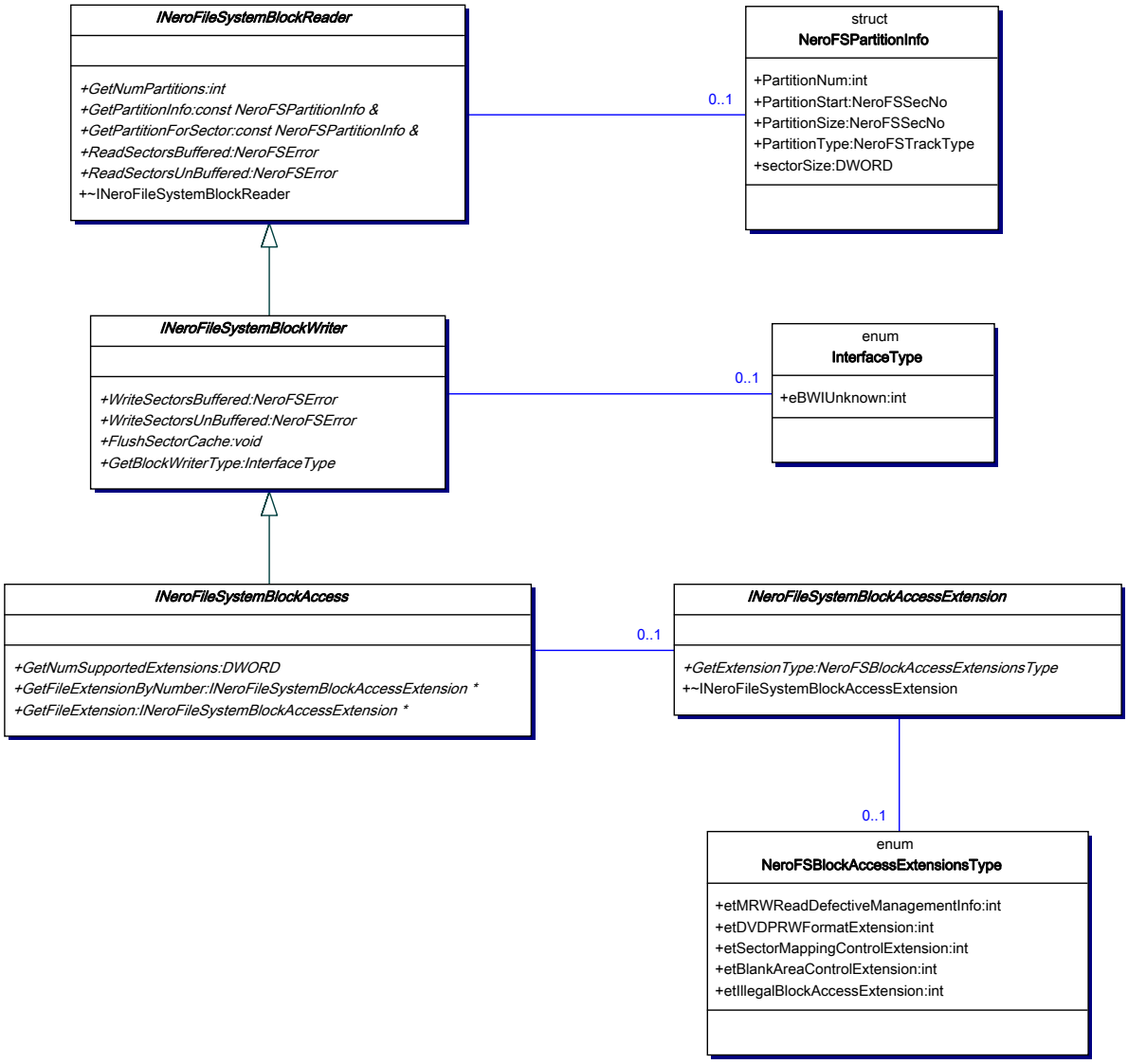
12.1.6. NeroGetSupportedAccessModesForDevice

This function will return a DWORD mask containing values as declared in enum AccessMode. Use ((result&eDesiredMode)!=0) to determine whether a specific mode is supported.

```
NEROAPI_API DWORD  NADLL_ATTR NeroGetSupportedAccessModesForDevice(
                                        NERO_DEVICEHANDLE aDeviceHandle);
```

12.2. File System Block Access Interface

These interfaces are part of the *NeroAPI* packet writing API. The packet writing API will return INeroFileSystemBlockAccess that can be used to have block access to a file system.



12.2.1. INeroFileSystemBlockAccess

This interface contains an extension scheme that will allow us to transparently extend the interface's functionality without losing binary compatibility.

```
class INeroFileSystemBlockAccess :public INeroFileSystemBlockWriter
{
public:
      virtual DWORD GetNumSupportedExtensions() = 0;
   virtual INeroFileSystemBlockAccessExtension
            *GetFileExtensionByNumber(int iNumExt) = 0;

      virtual INeroFileSystemBlockAccessExtension
            *GetFileExtension(NeroFSBlockAccessExtensionsType eExtType) =
0;
};
```

Description of members	
GetNumSupportedExtensions	Returns the number of supported extension fields.
GetFileExtensionByNumber	In combination with the method above, this method can be used to copy a set of extensions without knowing which ones are actually there. Extensions have to be passed through to any of the Nero file system generators, so it is essential to have some means of copying them.
GetFileExtension	Returns specified extension or NULL if not present.

12.2.2. INeroFileSystemBlockAccessExtension

All block device access extensions are derived from this one.

```
class INeroFileSystemBlockAccessExtension
{
public:
      virtual NeroFSBlockAccessExtensionsType GetExtensionType() const
                                                                      = 0;
      virtual ~INeroFileSystemBlockAccessExtension() {};
};
```

12.2.3. INeroFileSystemBlockReader

This is an abstract interface for reading from block devices. It will provide necessary data about the underlying medium as well as cache data if necessary.

```
class INeroFileSystemBlockReader
{
public:
  virtual int    GetNumPartitions() = 0;
  virtual const NeroFSPartitionInfo &GetPartitionInfo(int iNumPartition)
                                                                  = 0;

  virtual const NeroFSPartitionInfo &GetPartitionForSector(
                                                  NeroFSSecNo secNo) = 0;
  virtual NeroFSError ReadSectorsBuffered  (void *pData,
                                        NeroFSSecNo startSector,
                                        NeroFSSecNo noSectors,
                                        NeroFSSecNo &noSectorsRead)
                                                                  = 0;
  virtual NeroFSError ReadSectorsUnBuffered(void *pData,
                                           NeroFSSecNo startSector,
                                           NeroFSSecNo noSectors,
                                           NeroFSSecNo &noSectorsRead)
                                                                  = 0;
  virtual ~INeroFileSystemBlockReader() {};
};
```

Description of members	
GetNumPartitions	Retrieve the number of partitions.
GetPartitionInfo	Retrieve the partition information.
GetPartitionForSector	Returns the partition a given sector resides in.
ReadSectorsBuffered	The buffered reading method will use a cache to optimize file system access. It should be used when reading directory structures. This method returns error codes as described in NeroFSError. Your read requests may not cross partition boundaries!
ReadSectorsUnBuffered	The unbuffered reading method should be used when reading file contents. This method returns error codes as described in NeroFSError. Your read requests may not cross partition boundaries!

12.2.4. INeroFileSystemBlockWriter

The FileSystem block writer interface is derived from the block reader interface. It defines a path of access to RW file systems and partitions.

As is the case with the reader interface, the writer interface also provides two methods for sector access. While WriteSectorsUnBuffered will merely ensure the consistency of the read cache (write thru), WriteSectorsBuffered will not write anything to the block device immediately but will cache a certain amount of sectors before doing so.

The latter increases performance considerably but is prone to data loss in an unstable environment.

Please note that regardless of which method you use, you **must** call FlushSectorCache if you want all your data to be at their final physical location. The reason is that even when writing in UnBuffered mode, the driver may decide to not write away your data immediately. This depends on the underlying writing scheme (e.g. packet writing will always try to collect a certain amount of sectors).

```
class INeroFileSystemBlockWriter :public INeroFileSystemBlockReader
{
public:
      virtual NeroFSError WriteSectorsBuffered(const void *pData,
                                        NeroFSSecNo startSector,
                                        NeroFSSecNo noSectors,
                                        NeroFSSecNo &noSectorsWritten)
                                                                = 0;
      virtual NeroFSError WriteSectorsUnBuffered(const void *pData,
                                        NeroFSSecNo startSector,
                                        NeroFSSecNo noSectors,
                                        NeroFSSecNo &noSectorsWritten)
                                                                = 0;
      virtual void FlushSectorCache() = 0;

      virtual InterfaceType GetBlockWriterType() = 0;
};
```

Description of members	
WriteSectorsBuffered	Method for buffered writing.
WriteSectorsUnBuffered	Method for unbuffered writing.
FlushSectorCache	Force the flushing of the sector cache. FlushSectorCache will be performed implicitly upon deleting the block writer object.
GetBlockWriterType	Runtime type information to be used for downcasting into specialized interfaces.

12.2.5. InterfaceType

Specifies the block writer type. You can use this information to down-cast the interface to obtain specialized functionality.

No extensions to the normal block write interface are available so far.

```
enum InterfaceType
{
      eBWIUnknown
};
```

12.2.6. NeroFSBlockAccessExtensionsType

Type of an extension. Currently no extension is provided within the *NeroSDK*.

```
enum NeroFSBlockAccessExtensionsType
{
      etMRWReadDefectiveManagementInfo,
      etDVDPRWFormatExtension,
      etSectorMappingControlExtension,
      etBlankAreaControlExtension,
      etIllegalBlockAccessExtension,
      etHDPartitionInfo,
      etHDUsedBlockAccessExtention,
      etSectorPatchControlExtension,
      etHDPartitionLocker,
      etMediumCDStamp,
      etFragmentReservation,
      etUDFImageEditExtension,
      etSessionControl,
      etNeroFSBAExtensionReserved1,
      etNeroFSBAExtensionReserved2,
      etMediumInfo,
      etNeroFSBAExtensionReserved3,
      etNeroFSBAExtensionReserved4,
      etNeroFSBAExtensionReserved5,
      etNeroFSBAExtensionReserved6
};
```

12.2.7. NeroFSError

This enum is used to obtain the result of reading and writing operations.

```
typedef enum
{
      errOK=0,
      errEndOfDir,
      errEndOfFile,
      errReadError,
      errInvalidFS,
```

```
      errNoDirectory,
      errNoFile,
      errNotSupported,
      errIllegalArgument,
      errWriteError,
      errInternalError,
      errFileLocked
} NeroFSError;
```

Description of enumerators	
errOK	Operation successful.
errEndOfDir	Deprecated. Should never be returned, to be treated as errOK.
errEndOfFile	See the libc read command for reference.
errReadError	A read error has occurred.
errInvalidFS	The files system is not valid.
errNoDirectory	It has been attempted to perform a directory operation on an object that is no directory.
errNoFile	It has been attempted to perform a file operation on an object that is no file.
errNotSupported	Operation not supported.
errIllegalArgument	An illegal argument has been passed.
errWriteError	A write error has occurred.
errInternalError	An internal error has occurred.
errFileLocked	The file is locked.

12.2.8. NeroFSPartitionInfo

This struct stores information about a partition.

```
typedef struct
{
      int               PartitionNum;
      NeroFSSecNo       PartitionStart;
      NeroFSSecNo       PartitionSize;
      NeroFSTrackType   PartitionType;
      DWORD             sectorSize;
} NeroFSPartitionInfo;
```

Description of members	
PartitionNum	The current partition number.
PartitionStart	The start sector for this Partition.
PartitionSize	The number of sectors this Partition contains.
PartitionType	The type of Partition.
sectorSize	Sector size for this Partition.

12.2.9. NeroFSTrackType

Enumeration of file system track types.

```
typedef enum
{
      vtData=0,
      vtAudio
} NeroFSTrackType;
```

Description of enumerators	
vtData	Data Track.
vtAudio	Audio Track.

12.2.10. NeroFSSecNo

The sector number. All sector references use this type. LBA addressing is used throughout the interface.

```
typedef __int64 NeroFSSecNo;
```

13. Robot Control Interface

This interface is designed for associating duplication systems, also called “robots” or “robos“, with disc drives.

You can associate a robot with a drive by using the NeroAssociateRobo function with the device handle of a recorder.

13.1. NERO_COMMNODE_TYPE Enumeration

This enum type defines the character of the communication node, which can be either serial, parallel or USB.

```
typedef enum
{
   NCT_PORT_SERIAL,
   NCT_PORT_PARALLEL,
   NCT_PORT_USB
}
NERO_COMMNODE_TYPE;
```

13.2. NEROAPI_ROBO_ERROR Enumeration

This enum provides the supported error codes.

```
typedef enum
{
   NEROAPI_ROBO_OK               =0,
   NEROAPI_ROBO_NOTFOUND         =1,
   NEROAPI_ROBO_NOTSUPPORTED     =2
}
NEROAPI_ROBO_ERROR;
```

13.3. NeroPrintLabelCallback_t Callback

This callback is used for label printing. It will be called by the *NeroAPI* when it is time to print the label. The callback function must return TRUE if the label was printed successfully, FALSE otherwise.

This callback is called from **within** the NeroRobo driver thread. Therefore, usually no GUI interaction is allowed (spawning a new process is permitted, though). This callback must not return until the print job is done.

```
typedef BOOL(*NeroPrintLabelCallback_t)(void *data);
```

13.4. NERO_ROBO_DRIVER_INFO Structure

This struct provides information on one robot driver.

```
typedef struct
{
   char  nrdiIdentifier[256];
   DWORD nrdiVersionNumber;
   char  nrdiDLLName[256];
} NERO_ROBO_DRIVER_INFO;
```

13.5. NERO_ROBO_DRIVER_INFOS Structure

A list of robot driver information entries. The number of entries is determined by the nrdisNumDevInfos member.

```
typedef struct
{
   DWORD nrdisNumDevInfos;
   NERO_ROBO_DRIVER_INFO nrdiDevInfos[1];
} NERO_ROBO_DRIVER_INFOS;
```

13.6. NeroGetAvailableRoboDrivers Function

This function will return a list of available robot drivers.

```
NEROAPI_API NERO_ROBO_DRIVER_INFOS * NADLL_ATTR
                                          NeroGetAvailableRoboDrivers();
```

13.7. NeroAssociateRobo Function

Associate a robot of the specified type connected to a given port with a device handle.

For now, this function will always return NEROAPI_ROBO_OK as it does not actually do anything with the robot.

You can pass NULL instead of a print callback if printing shall be disabled.

```
NEROAPI_API NEROAPI_ROBO_ERROR NADLL_ATTR NeroAssociateRobo(
                                  NERO_DEVICEHANDLE aDeviceHandle,
                                  const char *robo_identifier,
                                  NERO_COMMNODE_TYPE port,
                                  int portnum,
                                  NeroPrintLabelCallback_t printcallback,
                                  void *printcallback_data);
```

13.8. NERO_ROBO_FLAG Enumeration

This enum is used when setting runtime options for the robot, i.e. settings that the robot will not remember beyond the current session. This enum is used as parameter to the NeroSetRoboFlag function.

```
typedef enum
{
   NERO_ROBO_FLAG_CLEANUP,
   NERO_ROBO_INSERTCD_RETRIES
} NERO_ROBO_FLAG;
```

Description of enumerators	
NERO_ROBO_FLAG_CLEANUP	Set this value in order to have the robot perform cleanup functions for itself. This might include removing any remaining discs from the trays etc. Note that the resulting behavior may sometimes look like a bad control flow. For example, the robot might try to remove discs when there are none. Therefore, you should not use this option unless you are cleaning up after a hard server crash or have another good reason to do so.
NERO_ROBO_INSERTCD_RETRIES	This flag specifies how often *Nero* is to try to insert another disc from the input tray if the disc in the drive is not writable. The current default value for the number of retries is 5, a value of 0 means trying indefinitely.

13.9. NeroSetRoboFlag Function

This function sets runtime options for the robot. Those will not be remembered beyond the current session.

With this function, the generic control flow of the robot can be set. The function below will return NEROAPI_ROBO_NOTSUPPORTED if a specific control flow option is not supported by the robot or not available for some other reason.

```
NEROAPI_API NEROAPI_ROBO_ERROR NADLL_ATTR NeroSetRoboFlag(
                                          NERO_DEVICEHANDLE aDeviceHandle,
                                          NERO_ROBO_FLAG  eRoboFlag,
                                          int iRoboValue);
```

14. Media Type Formats

14.1. Audio

The *NeroAPI* requires the use of PCM (Pulse Code Modulation), 44.1kHz, Stereo (left channel first), 16 bits per channel, Little Endian Word (Least Significant Byte first).

WAV and MP3 files can also be burnt on Audio-CD by passing their path.

14.2. Video

14.2.1. SVCD Creation with *Nero*

There has been some confusion about what kind of input files are accepted by *Nero* for VCD and SVCD. The general answer is:

MPEG files that have already been prepared for VCD or SVCD. If the files conform to the VCD or SVCD specs, *Nero* is able to write a VCD or SVCD on-the-fly without re-encoding the files.

It's important to realize that there are different types of MPEG files. What makes a MPEG suitable for a (S)VCD is way beyond the scope of this documentation text and has to be dealt with by the makers of MPEG encoders. It involves details and settings that simply cannot be chosen via the user interface of existing encoders, unless they have a button dedicated to "(S)VCD encoding".

Having said that, there is a way to at least make *Nero* happy with the source MPEG files. However, this is far from producing a standard compliant CD, because *Nero* cannot test all the relevant aspects.

The most obvious (and most easily met) requirement is picture size:

Format	PAL Resolution	NTSC Resolution
VCD, normal	352x288	352x240
VCD, high	704x576	704x480
SVCD, normal	480x576	480x480
SVCD, high	704x576	704x480

The "high" resolutions are only available for still images, not for movies. For still images, *Nero* will do the encoding by itself, so the picture sizes may differ. *Nero* will automatically fit the picture into the available space (in a future update, this will be user-configurable).

The frequency for the video is 25Hz for PAL and 29.97Hz for NTSC. The VCD format also allows a "MOVIE" resolution of 352x240 at 23.976Hz, but although this is legal, it is said to cause problems with some players.

Audio must be MPEG-1, layer 2, at 44.1kHz, stereo. SVCD also allows a second music channel and MPEG-2 multi-channel. The second channel is usually used for another language or – in case of Karaoke - for the music without the vocal track.

Apart from these obvious aspects, *Nero* also requires the MPEG-2 file to have a pack size that fits directly into a mode 2, form 2 block, i.e. it must be 2324 bytes large. If this (and for VCD also some other minor aspects) is not met, then *Nero* will list the file as having an "invalid stream encoding".

Nero does not test if scan information is stored in the user data of a stream. Scan information is required by the SVCD standard and might be required by certain players for seeking functions. *Nero* also accepts streams that contain invalid stream IDs.

15. FAQ

This is a collection of frequently asked questions from the *NeroSDK* forum and Nero AG's customer support.

We have tried to avoid duplicates in different sections. If you, for example, have a problem with multisession writing in combination with DVD media, you might want to scan both the "Multisession" and "DVD Issues" chapters.

15.1. NeroSDK License

What licensing requirements do I have to meet to use the *NeroSDK*?

You can find the license agreement in the file NeroSDK_License.txt in the *NeroSDK* package. To be able to work with the *NeroAPI* you additionally need a license of *Nero*. Additionally, all computers running your application need a valid *Nero* license.

Are there any limitations (for instance burn speed) if I use the *NeroSDK*?

The limitations depend on the *Nero* license you have installed on the computer.

Can I distribute the software that I made with *NeroSDK* as a part of commercial software?

Yes, you can. But on every computer that runs your software there will have to be a *Nero* version with a valid license installed. For details on the license please also read the file NeroSDK_license.txt that is contained in the *NeroSDK* package.

15.2. NeroSDK/NeroAPI Features

Can the *NeroSDK* create Slideshows of images in (S)VCD format?

Yes, the *NeroSDK* supports this.

Can the *NeroSDK* create Slideshow of images in DVD-Video format?

No, the *NeroSDK* does not support this.

Does the *NeroAPI* support writing Diskt@2?

The *NeroAPI* does not support this.

What recorders does the *NeroAPI* support?

Every recorder that is supported by *Nero* is also supported by the *NeroAPI*.

Can I copy a disc with the *NeroSDK*?

The *NeroSDK* does not offer functionality to copy a CD or DVD.

Can I use the trial version of *Nero 6* to get the *NeroAPI* to work?

Yes.

15.3. General Programming Issues

Where can I find the C interface to the *NeroAPI*?

You will find the C Interface to the *NeroAPI* in the NeroAPI.h file.

Where can I find good examples of how to use the *NeroAPI*?

NeroCmd utilizes much of the power of *NeroAPI*.

Is there any support for a Delphi interface?

Delphi will not be supported directly. The documentation will hopefully be good enough to enable you to write a wrapper.

Do I have to link to both NeroAPIGlueRT.lib and NeroAPIGlue.lib?

You should only link to one of the libraries. NeroAPIGlueRT.lib is used for dynamic linkage to the runtime library and NeroAPIGlue.lib for static linkage.

Can I use the NeroAPI with Borland C++ Builder?

Since *NeroSDK* 1.05, Borland C++ Builder is supported.

Please use the NeroAPIGlueBCPPB.lib library. You can find it in the Lib directory of the *NeroAPI* tree.

I have problems with Borland's C++ Builder. What can I do?

Upgrade to *NeroSDK* 1.05 which supports Borland's C++ Builder.

15.4. DVD Issues

How can I obtain a list of all devices supporting DVD read/write or DVD write?

Use NeroGetAvailableDrivesEx to get a list of devices in the NERO_SCSI_DEVICE_INFOS structure. Walk through the list and use the nsdiMediaSupport member of each NERO_SCSI_DEVICE_INFO structure to get the supported writable media of a drive. Use the nsdiMeadReadSupport member to get the media a drive is able to read.

To see if a drive is a recorder or a read-only device, you can evaluate the value of the nsdiDevType member.

Please do not use the mediaType parameter of NeroGetAvailableDrivesEx for this purpose. This parameter is only meant to obtain a list of available speeds for a certain media from the recorder.

Does the *NeroAPI* support writing multisession DVD?

Writing multisession on DVD-R and DVD+R is supported. For DVD+RW and DVD-RW virtual multisession (VMS) is supported by *NeroAPI* 6.0.0.12 or better (*NeroSDK* 1.04).

Virtual multisession is a technique that allows writing multisession discs on media types that do not support normal multisession, e.g. DVD-/+RW.

Virtual multisession media can be handled by the standard multisession methods of the *NeroAPI*. Only if you want to access a session that is not the last on the media, you have to use the special virtual multisession methods.

Warning: Many operating systems have problems when reading from DVD-R and DVD+R multisession media.

How can I detect the presence of a DVD media?

Use ncdiMediaType of struct NERO_CD_INFO instead of ncdiMediumType. ncdiMediumType is obsolete!

Why does NeroBurn not return while "Writing short lead-out" on DVD media?

Depending from the exact media type, writing the lead out might take a long time. If the amount of data to be written is relatively small, writing the lead-out portion appears to consume a disproportionate amount of time.

Especially, if the *NeroAPI* writes a high compatibility border to the disc, which means that at least 1 GB is written.

How do I get a list of VMS sessions written on DVD?

To determine if a media of type DVD+RW, DVD-RAM or DVD-RW is a virtual multisession media, check NERO_CD_INFO::ncdiMediumFlags against NCDIMF_VIRTUALMULTISESSION. Then, retrieve the session information using NeroGetVMSInfo.

You will need *NeroSDK* v1.04 and a recent *Nero* version.

I realized that I can burn DVD-Video using the *NeroAPI*. Does it allow creating and encoding Video-DVD projects, in other words does it include any of *Nero Vision Express 2* functionality?

No. You can only burn the generated files with the *NeroAPI* so that the generated DVD is standard compliant.

Please take a look at the *NeroVision API* which is contained in the *NeroSDK*. The *NeroVision API* offers a selected range of *Nero Vision Express 2* functionality.

I am trying to create a DVD-Video, but get an error 'Backup file 'VTS_01_0.BUP' should be identical to 'VTS_01_0.IFO'. What is wrong?

The .BUP files are the backup files of .IFO files.

For DVD-Video, the .BUP files must be identical to the .IFO files. In fact, only stand-alone DVD players need the .IFO files. However, .IFO files are so important for stand-alone DVD players that the backup files must exist. If the .BUP files differ from the .IFO files, the compilation is not valid for burning.

Does the *NeroAPI* support the burning of ISO images to DVD?

This became possible with version 6.3.0.6 of the *NeroAPI*.

Set nwiMediaType of struct NERO_WRITE_IMAGE to MEDIA_DVD_ANY.

Can I create a DVD-Video by providing the *NeroAPI* with .mpg or .avi files?

There is no special API for this. You have to prepare the files for the DVD-Video yourself. Then you can burn a UDF/ISO file system to DVD which has to contain all necessary files in the proper location.

For additional information, please consider the answer to the next question.

How can I create a DVD-Video with the *NeroAPI*?

For improved compatibility and standard compliance, use the following flags:

```
NBF_DVDP_BURN_30MM_AT_LEAST | NBF_BOOK_TYPE_DVDROM
```

The required file system flags are:

```
NCITEF_CREATE_ISO_FS | NCITEF_CREATE_UDF_FS |
NCITEF_DVDVIDEO_CMPT | NCITEF_DVDVIDEO_REALLOC
```

With *NeroAPI* 6.3.1.11, the Burn-at-once interface has become available. Burn-at-once offers functionality to burn a DVD-Video without creating the ISO file system on the hard disk first.

Why does a DVD+RW media stay open even if I configured the *NeroAPI* to finalize it?

DVD+RW can not be finalized in the usual meaning. And they also do not support real multisession like CD-R/RW.

The *NeroAPI* always adds the information that is required to continue DVD+RW.

(*Nero* will only allow this if the user did not choose "Finalize")

15.5. Video CD and Super Video CD

How do I instruct the *NeroAPI* to use PAL/NTSC format when creating a VCD/SVCD?

This is supported by Nero 6.0.0.17 (*NeroSDK* 1.04) and later versions.

The member nwvcdEncodingResolution within the struct NERO_WRITE_VIDEO_CD can be used to provide that information. Valid values are NERO_VIDEO_RESOLUTION_PAL and NERO_VIDEO_RESOLUTION_NTSC

How can I burn a VideoCD with data track with the *NeroAPI*?

You can use the IsoTrack property of the NERO_WRITE_VICEO_CD structure. It takes a CNeroIsoTrack class that can be created by NeroCreateIsoTrackEx or an implementation of your own. That class implementation would be derived from CNeroIsoTrack.

NeroCreateIsoTrackEx requires a NERO_ISO_ITEM that can be created by NeroCreateIsoItem.

15.6. Audio

Does the *NeroAPI* support ISRC (International Standard Recording Code)?

The *NeroAPI* can read ISRC information but cannot write it.

Please use NeroGetCDInfo to extract the ISRC. The information is contained in the struct NERO_TRACK_INFO in ntiISRC (only if NGCDI_READ_ISRC is present).

Is there any support of jitter correction by the *NeroAPI*?

No, the *NeroAPI* does not offer any jitter correction functionality. Only the jitter correction built into the devices will be used.

Why does burning Audio CDs with indexes result in an "Invalid field in parameter" error?

The *NeroAPI* expects the indexes as a multiple of the block size (2352). The description in the NeroAPI.h was a bit misleading in earlier versions and has been improved as of *NeroSDK* version 1.04.

15.7. Multisession

How can I prevent closing a disc, e.g. to start or continue a multisession disc?

Use the flag NBF_CLOSE_SESSION with the NeroBurn function.

During the importing of a track *Nero Burning ROM* reports "The Track you selected was not created using Nero's multisession option, References to you local files cannot be automatically restored" for a multisession disc started with *NeroAPI*. What does that mean?

The message just indicates that the multisession disc was not written by *Nero Burning ROM / Nero Express*. *Nero Burning ROM* writes additional information on multisession discs which allows it to offer some options like updating files on the disc if the content has changed, etc. The *NeroAPI* does not write this kind information. But you can nevertheless continue a multisession disc written with the *NeroAPI* with *Nero Burning ROM*.

Do I have to import the last or all tracks of a disc?

You normally import only one session. Which one, really depends on your particular requirements. In most cases it is the last.

When burning, the *NeroAPI* will rename a directory or file if the same name exists in a previous session. How can this be anticipated?

You can obtain the content of the previous session with NeroImportDataTrack. You will get an ISO item tree, which contains all files of the imported session.

To prevent the renaming of ISO items, make sure there are no duplicate item names. If a directory of the desired name already exists, arrange the new ISO items below the existing directory.

Is there any sample code to burn, import and continue a multisession data CD?

Yes, please have a look at the *NeroCMD* code. Importing is done in IsoTrack.cpp with NeroImportIsoTrackEx. To continue a multisession disc you just have to import the session you want to continue and add the resulting NERO_ISO_ITEM to the root of your NERO_ISO_ITEM tree.

To burn the disc with the possibility to later continue it, add the flag NBF_CLOSE_SESSION to dwFlags in NeroBurn.

How do I continue a multisession disc if I want to use my own CNeroIsoTrack implementation?

NeroImportDataTrack does not return a CIsoTrack but a tree of NERO_ISO_ITEMS which contain some information about the files on the disc. So what you actually have to do is to provide this information to your ISO track implementation and feed it to the *NeroAPI* when asked for it.

You should enhance the demo ISO track of NeroAPITest a bit so that it actually supports multisession. The information you have to take from the ISO items is:

```
// Used to reference a file from a previous session
long dataStartSec;
__int64 dataLength;
struct tm entryTime;
struct CImportInfo *importinfo;
```

Eventually, the information needs to be passed to the *NeroAPI* with these methods of CNeroIsoEntry:

```
// Can be used to reference files from previous session
virtual DWORD GetDataStartSec(); //
virtual BOOL IsDataFixed(); // return TRUE here
virtual BOOL GetEntryTime(struct tm *tm)
virtual CImportInfo *GetImportInfo() const;
```

What is the purpose of the CD stamp?

The purpose of the CD stamp is to let the *NeroAPI* request exactly the disc that the CD stamp describes. E.g. for multisession discs, you would not want the user to insert any disc but the one that contains the session from.

How can I write multiple sessions to a disc?

1. Use NeroGetCDInfo to check how many sessions are already on the media.
2. Use NeroImportDataTrack/NeroImportIsoTrackEx specifying the last session as session to import.
3. Append your new ISO items to the ISO item tree
4. Burn the resulting ISO item tree.

15.8. Size Information, Calculation and Estimation

How can I obtain the available size of a media for each media type?

Use NERO_CD_INFO::ncdiFreeCapacityInBlocks. One block has the size of 2048 bytes for data. For Audio and VCD/SVCD the size is 2352.

spaceAvaiable = (NeroCDInfo.FreeCapacityInBlocks()*_blockSize).

Why does NeroEstimateTrackSize() always return -1 with *NeroAPI* 6.0.0.0?

You need at least version 6.0.0.14 in order for NeroEstimateTrackSize to work. Please update your installation and retry.

How can I determine free disc space available for writing and used disc space with the NERO_CD_INFO structure returned by NeroGetCDInfo?

1. NERO_CD_INFO.ncdiFreeCapacityInBlocks should give you the information (blocks are 2048 bytes long)
2. Parse the track info array starting at NERO_CD_INFO.ncdiTrackInfos and add all NERO_TRACK_INFO.ntiTrackLengthInBlks.
   - If it contains "normal" data, the size is 2048 bytes.
   - If you put audio or video in these tracks (mode 2 form 2), the effective block size will be 2324 bytes.

Actually, the total size of a block is exactly 2352 bytes. Depending on the format, bytes are used for sync, header, EDC and ECC. In mode 1 and mode 2 form 1, you get 2048 for user data. In mode 2 form 2 there is no ECC, so you get 276 bytes more, but there are tradeoffs in reliability. Therefore, these kinds of tracks are typically used for audio and video.

What is the size limitation for an UDF volume name?

Currently, the *NeroAPI* limits the UDF volume name to 16 characters.

15.9. Packet Writing

Can I format a disc with UDF?

It is not possible to format a disc with the *NeroAPI*.

Does the Packet Writing interface of the *NeroAPI* offer the same functionality as *InCD*?

No.

Do I have to install *InCD* if I want to use the Packet Writing interface of the *NeroAPI*?

No.

Can I write/erase BYTE data on UDF formatted discs similar to a hard disk with the *NeroAPI*?

You can read and write sectors.

What is the scope of *NeroAPI* Packet Writing?

NeroAPI packet writing support is limited to

- reading sectors for CD-R media.
- read/write access to sector of CD-RW media already formatted in packet writing mode. This requirement also applies to DVD-RW media. DVD+RW media can be written to without prior formatting.

Anything required besides low-level read/write access (like file system drivers) needs to be developed by you since the current version of the *NeroAPI* does not provide access to packet writing media on a file system level.

NeroAPI Packet Writing will always overwrite sectors that already exist on the media. The *NeroAPI* does not provide support to format a packet writing medium, neither physically nor logically.

When using UDF, in which format will the disc be written?

The *NeroAPI* will write UDF 1.02, physical partition. Other formats are currently not supported.

15.10. Concurrency

Does the *NeroAPI* support writing to several recorders simultaneously?

The *NeroAPI* does not support this.

Does the *NeroAPI* support burning to several drives in parallel?

You cannot initiate two burns at the same time in the same process with *NeroAPI* but it should not be a problem to start two processes. Try burning with two separate instances of *NeroCMD*.

Is NeroBurn a synchronous or asynchronous function?

NeroBurn returns only after burning has been finished.

Can I lock the recorder for burning to prevent other users from ejecting the disc?

There is no such method. The *NeroAPI* locks the recorder by itself during burning.

15.11. Miscellaneous

What is the meaning of "using interface version" in the log file?

This is the version of the *NeroAPI* which was used for compiling the application, i.e. the version of the *NeroAPI* headers in the used *NeroSDK*.

Why does using the flag NBF_DISABLE_ABORT result in an exception?

The disable abort callback (NERO_PROGRESS::npDisableAbortCallback) is not set. This was a bug in the *NeroAPI* and has been fixed with *NeroAPI* 6.3.1.1.

How can I use GetDrives to obtain a list of recorders?

GetDrives always returns all drives available. You can check if a drive is a recorder, by looking at the DevType property of NeroDrive. It should be NERO_SCSI_DEVTYPE_WORM. You can also check which media are supported for writing by checking the MediaSupport property of a NeroDrive.

To check if a drive is an image recorder, test the Capabilities property of NeroDrive for NERO_CAP_IMAGE_RECORDER.

NeroBurn returns with NEROAPI_BURN_ERROR and GetLastError tells me "Initialization failed". What happened?

There a number of potential problems:

1. The last parameter of NeroBurn must not be empty! A NERO_PROGRESS structure is required there. Of course, you can set most of the callback functions to NULL in that structure.
2. You set nwcdNumTracks to 1, but you want to burn only the iso track and no audio tracks. nwcdNumTracks refers to the number of audio tracks.
3. You set nwcdMediaType to MEDIA_DVD_ANY|MEDIA_CDRW. This is not supported. You have to set it to a single media type like MEDIA_DVD_ANY or MEDIA_CD.
4. One of the tracks you are trying to burn does not exist.

Is there any way to burn a data disc using the C programming language?

You should be able to create a data disc in C by creating a tree of IsoItems and then create the necessary CIsoTrack pointer with NeroCreateIsoTrackEx.

You do not have to include NeroIsoTrack.h if you are using the NeroCreateIsoTrackEx method. Just forward the pointer you get from the NeroCreateIsoTrackEx method to the *NeroAPI*, e.g. by setting the nwcdIsoTrack member of the NERO_WRITE_CD struct.

How do I create a CD that is Windows readable with long file names?

Use all of the following options:

NERO_BURN_OPTION_CREATE_ISO_FS

NERO_BURN_OPTION_RELAX_JOLIET

NERO_BURN_OPTION_USE_JOLIET

Why can I not change the structure of a CD/DVD on the fly?

The file system with references to the file data needs to be written first, so the structure of the media must be known beforehand.

How do I write a disc using the FileSystemContent interface?

```
FileSystemContent::IFileSystemDescContainer
pContainer(NeroCreateFileSystemContainer(NULL));

// set the volume name
pContainer->SetName("Volume");
// Get the root directory container,
// build its content recursively
FileSystemContent::IDirectoryContainer *pRoot=pContainer->Root();

// add some files to the container
FileSystemContent::IDirectoryEntryContainer *pDirEntry= pRoot->
AddFile("autoexec.bat","c:\autoexec.bat",-1,-1);
pDirEntry=pRoot->AddFile("test.txt","c:\config.sys",-1,-1);

// add a directory to the container
FileSystemContent::IDirectoryContainer * pSubDirectory = pRoot-
>AddDirectory("New folder", -1);

// now you can use pSubDirectory to build the content of the sub directory
pDirEntry=pSubDirectory->AddFile("test.txt","c:\config.sys",-1,-1);
```

After you have built the file system tree, you can burn it with NeroBurn by using the NERO_WRITE_FILE_SYSTEM_CONTENT structure.

Is there a way of timing out so that NeroBurn returns with an error if the CD has not been inserted after a certain period?

No. You can add your own timer and if the timeout is reached, return FALSE in the idle callback, which will make the *NeroAPI* cancel the burning operation.

Why is my data CD empty?

```
*ppIsoTrack = NeroCreateIsoTrackEx (NULL, m_currentCDName, dwFlags );
```

You have to give a pointer to the root of you NERO_ISO_ITEM tree as first argument. Since you specify NULL, your CD will be empty.

Can the *NeroAPI* handle .nri files?

Files with .nri extension are proprietary project files for ISO compilations, either created by *Nero Burning ROM* or *Nero Express*.

Files of this kind, as well as other proprietary compilation files, such as .nra (audio disc compilations) or .nrv (video disc compilations), can not be read or created by the *NeroAPI* or other *NeroSDK* components.

Files with an .nrg extension, however, are disc images. This kind of image is created when a compilation is burnt to the Image Recorder. Files with .nrg extension can be read and created with the *NeroAPI*.

The NeroFiddles example can burn one file to a disc. How can I burn more than one file?

To write several files, you have to build a tree of NERO_ISO_ITEMs that represent the files/directories you want to burn on the disc.

Every NERO_ISO_ITEM has a member nextItem that points to the next file in the current directory. If the item is a subdirectory, the member subDirFirstItem points to the first file.

The items can be created by the NeroCreateIsoItem function. Then pass the NERO_ISO_ITEM pointer of the root item to NeroCreateIsoTrackEx.

The memory has to be freed after the burning process, for example by using the NeroFreeIsoItemTree function.

How can I burn data that I receive from a network directly from RAM to disc without creating temporary files first?

Please have a look at the NeroAPITest example of the *NeroSDK*.

In NeroIsoTrack.h you should study CDemoIsoHandle::GetFileName. Here you can see how to inform the *NeroAPI* that you want to provide the data with callbacks.

The FileSystemContent interface also allows writing on the fly.

With NERO_ISO_ITEM you can only write files present on your hard disk.

Why is the longFileName of NERO_ISO_ITEM a char* while longSourceFilePath is a const char*?

The reason is, that longFileName is allocated and freed by the *NeroAPI* if the item is a reference.

How do I create an ISO image of files on a hard drive?

Once you have your data prepared, call NeroBurn using an image recorder's device handle.

When your user dialog callback eventually receives the DLG_FILESEL_IMAGE event, supply the *NeroAPI* with a destination file name.

How can I use more than 64 characters with Joliet?

NeroSDK 1.03 and more recent versions support relaxing Joliet restrictions.

When providing the NCITEF_RELAX_JOLIET flag for the NeroCreateIsoTrackEx function, 109 characters can be used.

Can I use media type collections in all *NeroAPI* methods?

No. The *NeroAPI* provides two different method parameter types for media:

- NERO_MEDIA_TYPE, which describes one particular media type
- NERO_MEDIA_SET, which represents a set of several media types

Media type collections are usually created with the help of the bitwise OR operator '|'. Only if the parameter is of type NERO_MEDIA_SET, it becomes legal to pass media type collections to the method in question.

Otherwise, you may only pass one single media type, as defined in the NeroAPI.h header file.

16. Known Limitations

- Currently, there are no Linux versions of the *NeroAPI* and the *Nero* Software Development Kit (*NeroSDK*).
- Methods of the *NeroAPI* should not be called simultaneously from different threads.
- Only one recorder can be accessed at a time.

17. Bibliography

17.1. C Programming Books

For those who never have programmed before:

Greg M. Perry: Absolute Beginner's Guide to C

From the guys who invented C. Only for beginners who can appreciate a challenge:

Brian W. Kernighan and Dennis M. Ritchie: The C Programming Language

17.2. C Programming Online Resources

This site is both for C and C++ programming

http://www.cprogramming.com

Steve Summit's Introductory C course

http://www.eskimo.com/~scs/cclass/cclass.html

17.3. C++ Programming Books

If you know something better than these, let us know.

Ivor Horten: Beginning Visual C++ 6

Davis Chapman: Sams Teach Yourself Visual C++ in 21 days

From the inventor of C++. Not for the faint of heart.

Bjarne Stroustrup: The C++ Programming Language

The author's favorite author. ;-) Read it if you think you know C++ inside and out.

James Coplien: Advanced C++ Programming Styles and Idioms

17.4. C++ Online Resources

Valencia Community College C++ programming course

http://m2tech.net/cppclass/

Intended for C users who want to make the transition to C++

http://www.icce.rug.nl/docs/cplusplus/cplusplus.html

A very good site

http://www.codeproject.com

17.5. General CD/CD-ROM Online Resources

Glossary of CD-ROM and DVD technologies

http://www.sigcat.org/resource/gloss697.htm

17.6. Audio CD Online Resources

Digital Audio on CD

http://www.disctronics.co.uk/cdref/cdaudio/cdaudio.htm

17.7. Super Video CD Online Resources

MPEG-2 encoder test

http://www.tecoltd.com/enctest/enctest.htm

A well-researched page on SVCD

http://www.iki.fi/znark/video/svcd/overview/

German page with a similar mission

http://www.ratos.de/