NeroROBO SDK

V1.0.0.0

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

Nero AG, Im Stoeckmaedle 18, 76307 Karlsbad, Germany

1. Contents

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

In recent years, duplication systems have become increasingly popular.

Nero AG was only able to provide drivers for a limited number of systems. Therefore, we decided to add the *NeroROBO SDK* to the already existing *NeroSDK*.

This SDK will empower duplication system manufacturers or end users to implement their own driver that will work with *NeroROBO*.

3.2. Overview

In this document we will explain how to implement your own robot drivers, which will be well recognized by *Nero*. We introduce the classes and interfaces, which are used by *Nero* to correctly recognize valid robot drivers and utilize their functionality.

Of course you may purchase drivers for already supported duplication systems. If you are interested in purchasing *Nero* robot drivers - please send us an e-mail, we will contact you as soon as possible:

sales-nerorobo@nero.com

3.3. Requirements

The *NeroROBO SDK* will work on any platform which is fit for hosting ***Nero 6.0.0.0***. *Nero* needs to be installed and licensed prior to using *NeroROBO*.

You can obtain the latest version of *Nero* from http://www.nero.com.

Additionally, the **NeroRobo.dll** file, which is included in the *NeroSDK* package, needs to be copied to *Nero's* installation directory.

3.4. Required Skills

This documentation is directed towards software developers who have gathered some experience in programming C++.

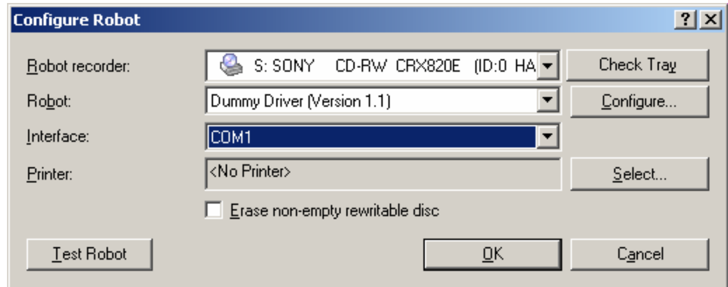
4. The Include Directory

The include directory of the *NeroROBO SDK* contains the header files that are required for implementing a robot driver plug-in that is compatible with *Nero*.

File	Description
AbstractConfigurationStorage.h	Interface for saving/querying robot configuration settings. This file describes an abstract interface to a settings management feature for external modules.
AbstractUserDialog.h	Interface for Graphical User Interface. This file defines an abstract way to represent a dialog with an interface exported to e.g. a DLL or a separate thread.
CCOMSerial.h	Interface for serial (COM) port communication. This is an extension to the CSerial interface to allow COM Port specific configuration for interfaces returning the type PORT_SERIAL at runtime.
CSerial.h	Interface for different serial ports. The file describes a compatibility class for serial port access. *Note that 'serial port' refers to all interfaces like USB, COM, LPT receiving data sequentially.* This class is pure virtual.
GeneralPluginDLL.h	Major *Nero* plug-in interface This is the General *Nero* Plug-in DLL Interface definition file.
NeroRoboFirmwareCheck.h	Interface for robot recognition. Class CNeroRoboFirmwareCheck defines an additional robot driver interface, which enables *Nero* to determine the firmware of the connected robot. This interface is used by *Nero* mainly to automatically recognize robots if particular a port was selected in the configuration dialog.
RoboCommNode.h	Interface for robot communication node. This file describes the robot communication node interface that is passed to each robot driver upon initialization. A robot driver can use this communication node to communicate with its hardware, the user and, in some limited ways, the operating system.
RoboDriver.h	Interface of robot driver. This file describes the interface each NeroROBO driver has to implement to be accessible from *Nero*.

5. DummyDriver Example

The DummyDriver is an example implementation of a robot driver. Whenever a real robot would perform a physical action, the dummy robot will display a message box for a certain amount of time.



You can use the DummyDriver example as skeleton for your own drivers that are actually are capable of controlling a robot. The driver is contained within a DLL which meets *Nero's* general requirements for Plug-in DLLs.

5.1. Globals

Global functions are implemented in DummyDriverDLL.cpp.

```
#include "DummyDriverConfig.h"
#include "DummyDriverFirmwareCheck.h"

BOOL APIENTRY DllMain(    HANDLE hModule, DWORD  ul_reason_for_call,
                          LPVOID lpReserved)
{
   return TRUE;
};
```

5.1.1. NERO_PLUGIN_GetPluginName

This function merely returns the ID string of the robot. DUMMYDRIVER_ID is defined as "Dummy Driver" in DummyDriver.h.

```
const char *NERO_PLUGIN_GetPluginName()
{
   return DUMMYDRIVER_ID;
};
```

5.1.2. NERO_PLUGIN_GetVersion

This function returns version information for the robot. The version is also defined in DummyDriver.h.

```
DWORD NERO_PLUGIN_GetVersion()
{
   return (DUMMYDRIVER_MAJOR<<16)+DUMMYDRIVER_MINOR;
};
```

5.1.3. NERO_PLUGIN_GetInterfaceVersion

This function reports the NeroROBO SDK interface version the driver is compatible with. The ROBODRIVER_CURRENTVERSION constant is defined in RoboDriver.h which is part of the distribution.

```
DWORD NERO_PLUGIN_GetInterfaceVersion()
{
   return ROBODRIVER_CURRENTVERSION;
};
```

5.1.4. NERO_PLUGIN_GetPluginType

This function returns the type of plug-in the DLL implements. ROBODRIVER_PLUGINTYPE is defined in RoboDriver.h.

```
const char *NERO_PLUGIN_GetPluginType()
{
 return ROBODRIVER_PLUGINTYPE;
};
```

5.1.5. NERO_PLUGIN_SetVendorInformations

This function is used by a caller to set vendor information. This information is not required. We do nothing in the implementation part.

```
void NERO_PLUGIN_SetVendorInformations(const char *szCompany,
                                            const char *szProductName)
{
};
```

5.1.6. nerorobodriver_createdriver

Create an instance of the CDummyRoboDriver class and pass it back to the caller.

```
CNeroRoboDriver *nerorobodriver_createdriver(IRoboCommNode *commnode)
{
   return new CDummyRoboDriver(commnode);
};
```

5.1.7. nerorobodriver_configuredriver

Configure the robot driver, using the IUserDialogFactory that was passed to the function as parameter.

```
void nerorobodriver_configuredriver(IUserDialogFactory *pDialogFactory)
{
 CDummyRoboDriverConfig dummyConfig(pDialogFactory);
 dummyConfig.Exec();
}
```

5.1.8. nerorobodriverV2_createdriver

This function is a more recent version of the createdriver function. It accepts an additional recorder name, which we ignore in this case.

```
CNeroRoboDriver *nerorobodriverV2_createdriver(IRoboCommNode *pCommNode,
const char *pcRecorderName)
{
 return nerorobodriver_createdriver(pCommNode);
}
```

5.1.9. nerorobodriverV2_configuredriver

Similar to the V2_createdriver function, we accept an additional parameter, which is not used by our implementation.

```
void nerorobodriverV2_configuredriver(IUserDialogFactory
                          *pDialogFactory, const char *pcRecorderName)
{
 nerorobodriver_configuredriver(pDialogFactory);
}
```

5.1.10. nerorobodriver_firmwarecheck

Return an instance of the CDummyRoboDriverFirmwareCheck class which can be used by the caller to obtain the required information.

```
CNeroRoboFirmwareCheck *nerorobodriver_firmwarecheck(CSerial
                                                          *pInterface)
{
 return new CDummyRoboDriverFirmwareCheck(pInterface);
}
```

5.2. Definitions in DummyDriver.h

This header file contains a number of defines and a class that implements a dummy robot driver using *Nero's* recent robot driver scheme.

The following include directives are required to get access to the CNeroRoboDriver class and the IRoboCommNode interface.

```
#include "RoboDriver.h"
#include "RoboCommNode.h"
```

Our driver name which will be presented to the application.

```
#define DUMMYDRIVER_ID    "Dummy Driver"
```

And the driver version number, which we have defined as 1.1.

```
#define DUMMYDRIVER_MAJOR 1
#define DUMMYDRIVER_MINOR 1
```

Simulated robot movement for 5 seconds to provide delays, which are far from being realistic.

```
#define SIMULATEROBOMOVEMENT_TIMEOUT 5000
```

By default we should print after burning and parallelize these two processes.

```
#define DEFAULTCONFIG_PRINTBEFOREBURN   FALSE
#define DEFAULTCONFIG_PARALLELIZE       TRUE
#define CONFIG_PRINTBEFOREBURN          "PrintBeforeBurn"
#define CONFIG_PARALLELIZE              "Parallelize"
```

5.3. CDummyRoboDriver Class

This class is derived from CNeroRoboDriver.

```
class CDummyRoboDriver : public CNeroRoboDriver
{
public:
 virtual ~CDummyRoboDriver();
  ...
};
```

This class implements the driver's interface towards the *Nero* core application. (Think of hooks here.)

The public methods that are listed below will be called

- at the beginning of a burning session, just after the driver has been loaded
- when *Nero* wants the robot to insert a writeable disc
- when *Nero* has found an invalid disc in the recorder and wants it to be removed
- when the recording process finished successfully
- when the recording process finished with an error

```
public:
  virtual BOOL InitBurn();
 virtual BOOL PreBurn();
 virtual BOOL InsertCD();
 virtual BOOL RemoveNonWriteableCD();
 virtual BOOL RemoveUntouchedCD();
 virtual BOOL PostBurnOK();
 virtual BOOL PostBurnFAILURE();
 virtual BOOL ExitBurn();
 virtual BOOL CleanUp();
 virtual BOOL StartPrintLabel();
 virtual BOOL StartRoboTest();
```

Only this global function is allowed to call the class constructor.

```
 friend CNeroRoboDriver *nerorobodriver_createdriver(
                                              IRoboCommNode *commnode);
```

Private constructor, called from nerorobodriver_createdriver().

```
private:
 CDummyRoboDriver(IRoboCommNode *commnode);
```

Print thread handling methods:

```
 static UINT PrintThreadEntry(void *pData);
 UINT PrintThread();
 BOOL WaitForPrintThread();
```

Private data:

```
 IRoboCommNode*    m_pCommNode;
 IRoboMutex*       m_pRoboMutex;
 IRoboThread*      m_pPrintThread;
 CSerial*          m_pCommInterface;
 BOOL              m_bShowInfoMessage;
 BOOL              m_bPrintBeforeBurn;
 BOOL              m_bParallelizePrinting;
 BOOL              m_bIsTesting;
 BOOL              m_bHasNewInterface;

 IRoboCommNode::ERoboLocationNode    m_eCurrentDiscPos;
```

Data member	Description
m_pCommNode	Our access to hardware, the user and the operating system.
m_pRoboMutex	Synchronize between print and burn threads.
m_pPrintThread	To illustrate parallelization, we print in separate thread.
m_pCommInterface	Communication interface to the robot.
m_bShowInfoMessage	Initially TRUE; FALSE if user wishes to turn off the display of information dialogs.
m_bPrintBeforeBurn	Should we print before burning?
m_bParallelizePrinting	If TRUE, we will burn and print at the same time. This speeds up up the burning/printing process.
m_bIsTesting	We are only testing.
m_bHasNewInterface	Important if working with older *Nero* versions.
m_eCurrentDiscPos	Remember current disc position to perform appropriate cleaning tasks.

5.3.1. Constructor

The constructor performs basic initialization and reads persistent settings from the registry.

```
CDummyRoboDriver::CDummyRoboDriver(IRoboCommNode *commnode)
{
```

Initialize our members.

```
 m_pCommNode              = commnode;
 m_pRoboMutex             = (m_pCommNode != NULL) ?
        m_pCommNode->CreateMutex() : NULL;
 m_pPrintThread           = NULL;
```

```
m_pCommInterface          = (m_pCommNode != NULL) ?
       m_pCommNode->GetSerialInterface() : NULL;
m_eCurrentDiscPos         = IRoboCommNode::ERLN_INPUT;
m_bShowInfoMessage        = TRUE;
```

Retrieve print setting from the Registry.

```
m_bPrintBeforeBurn        = (m_pCommNode != NULL) ?
       m_pCommNode->GetConfigurationValueBOOL(CONFIG_PRINTBEFOREBURN,
       DEFAULTCONFIG_PRINTBEFOREBURN):DEFAULTCONFIG_PRINTBEFOREBURN;
m_bParallelizePrinting    = (m_pCommNode != NULL) ?
       m_pCommNode->GetConfigurationValueBOOL(
       CONFIG_PARALLELIZE, DEFAULTCONFIG_PARALLELIZE):
                                    DEFAULTCONFIG_PARALLELIZE;
m_bIsTesting              = FALSE;
```

Configure the COM interface, although we do not use it here.

```
if((m_pCommInterface != NULL) &&
                      (m_pCommInterface->GetPortType() == PORT_SERIAL))
   static_cast<CCOMSerial*>(m_pCommInterface)->SetConfiguration(
           14400, 8, CCOMSerial::CPARITY_NO, CCOMSerial::STOPBITS_ONE);
```

Beginning with interface version 4, we can use some new communication node methods.

```
m_bHasNewInterface = (NERO_PLUGIN_GetInterfaceVersion() >= 4);
```

We do not parallelize if printing should be done before burning.

```
if(m_bPrintBeforeBurn)
    m_bParallelizePrinting = FALSE;
}
```

5.3.2. Destructor

In the destructor, we wait for the print thread before we destroy our driver to be sure the robot is idle.

```
CDummyRoboDriver::~CDummyRoboDriver()
{
 WaitForPrintThread();
 if(m_pRoboMutex != NULL)
     delete m_pRoboMutex, m_pRoboMutex = NULL;
}
```

5.3.3. InitBurn

This function is called at the beginning of a recording session; right after the driver has been loaded. Depending on the value of m_bShowInforMessage, a descriptive text will be displayed or not. This method will only be called once.

```
BOOL CDummyRoboDriver::InitBurn()
{
 BOOL bResult = TRUE;
```

```
 if(m_bShowInfoMessage)
 {
    CUserDialogHandle *pErrorMessage =
        m_pCommNode->CreateMessage(  "InitBurn is called.\n" \
        "Do some INITIALIZATION like RESETING robot here.\n" \
        "Press Cancel, so these messages will not be displayed
                             anymore.", ROBOCOMMNODE_MSG_HINT);
    if(pErrorMessage != NULL) pErrorMessage->WaitForUserFeedback();
    m_bShowInfoMessage = (pErrorMessage->GetFeedback() !=
                             CUserDialogHandle::DLGRESULT_CANCEL);
    if(pErrorMessage != NULL)
        delete pErrorMessage, pErrorMessage = NULL;
 }
 return bResult;
}
```

5.3.4. PreBurn

This method will be called in every recording cycle, right before burning begins; this is on of the potential places to perform label printing.

```
BOOL CDummyRoboDriver::PreBurn()
{
 BOOL bResult = TRUE;

 if((m_pCommNode != NULL) && (m_bShowInfoMessage))
 {
    CUserDialogHandle *pErrorMessage =
        m_pCommNode->CreateMessage(  "PreBurn is called.\n" \
        "Now you can PRINT label before burning it.\n" \
        "Press Cancel, so these messages will not be displayed
                             anymore.", ROBOCOMMNODE_MSG_HINT);
    if(pErrorMessage != NULL)
        pErrorMessage->WaitForUserFeedback();

    m_bShowInfoMessage = (pErrorMessage->GetFeedback() !=
                             CUserDialogHandle::DLGRESULT_CANCEL);

    if(pErrorMessage != NULL)
        delete pErrorMessage, pErrorMessage = NULL;
 }
 return bResult;
}
```

5.3.5. InsertCD

This method is called when *Nero* requires a disc in the tray.

```
BOOL CDummyRoboDriver::InsertCD()
{
 BOOL bResult = TRUE;

 if(m_pCommNode != NULL)
 {
     if(m_bShowInfoMessage)
     {
         CUserDialogHandle *pErrorMessage =
             m_pCommNode->CreateMessage(  "InsertCD is called.\n" \
             "Send move command to your robot.\n" \
             "MOVE the disc to RECORDER or PRINT it before\n" \
             "Press Cancel, so these messages will not be displayed
                                  anymore.", ROBOCOMMNODE_MSG_HINT);
         if(pErrorMessage != NULL)
             pErrorMessage->WaitForUserFeedback();

         m_bShowInfoMessage = (pErrorMessage->GetFeedback() !=
                                  CUserDialogHandle::DLGRESULT_CANCEL);

         if(pErrorMessage != NULL)
             delete pErrorMessage, pErrorMessage = NULL;
     }

     CUserDialogHandle *pDialog = NULL;
```

Check if printing label is possible.

```
     if(m_bPrintBeforeBurn && m_pCommNode->LabelPrintEnabled())
     {
         WaitForPrintThread();
         CN_LOCK;
         pDialog = m_pCommNode->CreateMoveCDDialog(
                 IRoboCommNode::ERLN_INPUT, IRoboCommNode::ERLN_PRINTER);
         CN_UNLOCK;
         m_pCommNode->SuspendThread(SIMULATEROBOMOVEMENT_TIMEOUT);
         CN_LOCK;
         if(pDialog != NULL)
             delete pDialog, pDialog = NULL;
         CN_UNLOCK;
```

If the new interface version is supported, the CommNode’s IsAbortedImmediate function is used to determine if the operation shall be stopped.

```
         m_eCurrentDiscPos = IRoboCommNode::ERLN_PRINTER;
```

```
        if(m_bHasNewInterface ? !m_pCommNode->IsAbortedImmediate():TRUE)
        {
            CN_LOCK;
            pDialog = m_pCommNode->CreatePrintCoverDialog();
            CN_UNLOCK;
            m_pCommNode->SuspendThread(SIMULATEROBOMOVEMENT_TIMEOUT);
            CN_LOCK;
            if(pDialog != NULL)
                delete pDialog, pDialog = NULL;
            CN_UNLOCK;

            if(m_bHasNewInterface ?
                                !m_pCommNode->IsAbortedImmediate() : TRUE)
                pDialog = m_pCommNode->CreateMoveCDDialog(
                                            IRoboCommNode::ERLN_PRINTER,
                                            IRoboCommNode::ERLN_RECORDER);
            else
                bResult = FALSE;
        }
        else
            bResult = FALSE;
    }
    else
    {
        CN_LOCK;
        pDialog = m_pCommNode->CreateMoveCDDialog(
                                            IRoboCommNode::ERLN_INPUT,
                                            IRoboCommNode::ERLN_RECORDER);
        CN_UNLOCK;
    }

    if(bResult)
    {
        m_pCommNode->SuspendThread(SIMULATEROBOMOVEMENT_TIMEOUT);
        m_eCurrentDiscPos = IRoboCommNode::ERLN_RECORDER;
        bResult           = m_bHasNewInterface ?
                                !m_pCommNode->IsAbortedImmediate() : TRUE;
    }

    CN_LOCK;
    if(pDialog != NULL)
        delete pDialog, pDialog = NULL;
    CN_UNLOCK;
 }
 else
    bResult = FALSE;

 return bResult;
}
```

5.3.6. RemoveNonWriteableCD

This function is called whenever a disc is not writeable.

```
BOOL CDummyRoboDriver::RemoveNonWriteableCD()
{
 BOOL bResult = TRUE;

 if(m_pCommNode != NULL)
 {
     if(m_bShowInfoMessage)
     {
         CUserDialogHandle *pErrorMessage =
             m_pCommNode->CreateMessage(
             "RemoveNonWriteableCD is called.\n" \
             "Send move command to your robot.\n" \
             "MOVE the disc to WASTE bin because it cannot be burned.\n"\
             "Press Cancel, so these messages will not be displayed
             anymore.", ROBOCOMMNODE_MSG_HINT);
         if(pErrorMessage != NULL)
             pErrorMessage->WaitForUserFeedback();

         m_bShowInfoMessage = (pErrorMessage->GetFeedback() !=
                                 CUserDialogHandle::DLGRESULT_CANCEL);

         if(pErrorMessage != NULL)
             delete pErrorMessage, pErrorMessage = NULL;
     }

     CN_LOCK;
     CUserDialogHandle *pDialog = m_pCommNode->CreateMoveCDDialog(
                                         IRoboCommNode::ERLN_RECORDER,
                                         IRoboCommNode::ERLN_WASTEBIN);
     CN_UNLOCK;
     m_pCommNode->SuspendThread(SIMULATEROBOMOVEMENT_TIMEOUT);

     CN_LOCK;
     if(pDialog != NULL)
         delete pDialog, pDialog = NULL;
     CN_UNLOCK;

     m_eCurrentDiscPos = IRoboCommNode::ERLN_WASTEBIN;
     bResult           = m_bHasNewInterface ?
                             !m_pCommNode->IsAbortedImmediate() : TRUE;
 }

 return bResult;
}
```

5.3.7. RemoveUntouchedCD

This function is called when *Nero* decides not to use a disc that is already in the tray.

```
BOOL CDummyRoboDriver::RemoveUntouchedCD()
{
 BOOL bResult = TRUE;

 if(m_pCommNode != NULL)
 {
     if(m_bShowInfoMessage)
     {
         CUserDialogHandle *pErrorMessage = m_pCommNode->CreateMessage(
             "RemoveUntouchedCD is called.\n" \
             "Send move command to your robot.\n" \
             "MOVE the disc back to INPUT bin because it was not
                                                       burned.\n" \
             "Press Cancel, so these messages will not be displayed
                                 anymore.", ROBOCOMMNODE_MSG_HINT);
         if(pErrorMessage != NULL)
             pErrorMessage->WaitForUserFeedback();

         m_bShowInfoMessage = (pErrorMessage->GetFeedback() !=
                             CUserDialogHandle::DLGRESULT_CANCEL);

         if(pErrorMessage != NULL)
             delete pErrorMessage, pErrorMessage = NULL;
     }

     CN_LOCK;
     CUserDialogHandle *pDialog = m_pCommNode->CreateMoveCDDialog(
                                         IRoboCommNode::ERLN_RECORDER,
                                         IRoboCommNode::ERLN_INPUT);
     CN_UNLOCK;
     m_pCommNode->SuspendThread(SIMULATEROBOMOVEMENT_TIMEOUT);

     CN_LOCK;
     if(pDialog != NULL)
         delete pDialog, pDialog = NULL;
     CN_UNLOCK;

     m_eCurrentDiscPos = IRoboCommNode::ERLN_INPUT;
     bResult           = m_bHasNewInterface ?
                           !m_pCommNode->IsAbortedImmediate() : TRUE;
 }

 return bResult;
}
```

5.3.8. PostBurnOK

The disc was recorded successfully. The tray is open when this method is called.

```
BOOL CDummyRoboDriver::PostBurnOK()
{
 BOOL bResult = TRUE;

 if(m_pCommNode != NULL)
 {
     if(m_bShowInfoMessage)
     {
         CUserDialogHandle *pErrorMessage = m_pCommNode->CreateMessage(
             "PostBurnOK is called.\n" \
             "Send move command to your robot.\n" \
             "MOVE the disc to OUTPUT bin or PRINT it before.\n" \
             "Press Cancel, so these messages will not be displayed
                                   anymore.", ROBOCOMMNODE_MSG_HINT);
         if(pErrorMessage != NULL)
             pErrorMessage->WaitForUserFeedback();

         m_bShowInfoMessage = (pErrorMessage->GetFeedback() !=
                                 CUserDialogHandle::DLGRESULT_CANCEL);

         if(pErrorMessage != NULL)
             delete pErrorMessage, pErrorMessage = NULL;
     }

     CUserDialogHandle *pDialog = NULL;

     if(!m_bPrintBeforeBurn && m_pCommNode->LabelPrintEnabled())
     {
         WaitForPrintThread();

         CN_LOCK;
         pDialog = m_pCommNode->CreateMoveCDDialog(
                                         IRoboCommNode::ERLN_RECORDER,
                                         IRoboCommNode::ERLN_PRINTER);
         CN_UNLOCK;
         m_pCommNode->SuspendThread(SIMULATEROBOMOVEMENT_TIMEOUT);
         CN_LOCK;
         if(pDialog != NULL)
             delete pDialog, pDialog = NULL;
         CN_UNLOCK;

         m_eCurrentDiscPos = IRoboCommNode::ERLN_PRINTER;
```

Create separate print thread, so the driver can perform another task (e.g. insert new disc in the recorder) during printing.

```
        if(m_bHasNewInterface ?
                            !m_pCommNode->IsAbortedImmediate() : TRUE)
            m_pPrintThread = m_pCommNode->CreateThread(
                                        PrintThreadEntry, (void*)this);
        else
            bResult = FALSE;

        if(!m_bParallelizePrinting /*|| m_bIsTesting*/)
            WaitForPrintThread();
    }
    else
    {
        CN_LOCK;
        pDialog = m_pCommNode->CreateMoveCDDialog(
                                        IRoboCommNode::ERLN_RECORDER,
                                        IRoboCommNode::ERLN_OUTPUT);
        CN_UNLOCK;
        m_pCommNode->SuspendThread(SIMULATEROBOMOVEMENT_TIMEOUT);
        m_eCurrentDiscPos = IRoboCommNode::ERLN_OUTPUT;
    }

    if(bResult)
        bResult= m_bHasNewInterface ?
                            !m_pCommNode->IsAbortedImmediate() : TRUE;

    CN_LOCK;
    if(pDialog != NULL)
        delete pDialog, pDialog = NULL;
    CN_UNLOCK;
 }

 return bResult;
}
```

5.3.9. PostBurnFAILURE

When this function is called, the recording process has terminated with an error. The tray is open.

```
BOOL CDummyRoboDriver::PostBurnFAILURE()
{
 BOOL bResult = TRUE;

 if(m_pCommNode != NULL)
 {
    if(m_bShowInfoMessage)
    {
        CUserDialogHandle *pErrorMessage = m_pCommNode->CreateMessage(
            "PostBurnFAILURE is called.\n" \
            "Send move command to your robot.\n" \
```

```
            "MOVE the disc to WASTE bin because burning failed.\n" \
            "Press Cancel, so these messages will not be displayed
                                    anymore.", ROBOCOMMNODE_MSG_HINT);
        if(pErrorMessage != NULL)
            pErrorMessage->WaitForUserFeedback();

        m_bShowInfoMessage = (pErrorMessage->GetFeedback() !=
                                CUserDialogHandle::DLGRESULT_CANCEL);

        if(pErrorMessage != NULL)
            delete pErrorMessage, pErrorMessage = NULL;
    }

    CN_LOCK;
    CUserDialogHandle *pDialog = m_pCommNode->CreateMoveCDDialog(
                                        IRoboCommNode::ERLN_RECORDER,
                                        IRoboCommNode::ERLN_WASTEBIN);
    CN_UNLOCK;
    m_pCommNode->SuspendThread(SIMULATEROBOMOVEMENT_TIMEOUT);

    CN_LOCK;
    if(pDialog != NULL)
        delete pDialog, pDialog = NULL;
    CN_UNLOCK;

    m_eCurrentDiscPos = IRoboCommNode::ERLN_WASTEBIN;
    bResult           = m_bHasNewInterface ?
                            !m_pCommNode->IsAbortedImmediate() : TRUE;
 }

 return bResult;
}
```

5.3.10. CleanUp

This function is only called once at the very end of a recording session, even if multiple discs were created.

```
BOOL CDummyRoboDriver::CleanUp()
{
 BOOL bResult = TRUE;

 if(m_bShowInfoMessage)
 {
    CUserDialogHandle *pErrorMessage = m_pCommNode->CreateMessage(
            "CleanUp is called.\n" \
            "REMOVE remaining discs from PRINTER and RECORDER.\n" \
            "Press Cancel, so these messages will not be displayed
                                    anymore.", ROBOCOMMNODE_MSG_HINT);
    if(pErrorMessage != NULL)
```

```
        pErrorMessage->WaitForUserFeedback();

    m_bShowInfoMessage = (pErrorMessage->GetFeedback() !=
                                CUserDialogHandle::DLGRESULT_CANCEL);

    if(pErrorMessage != NULL)
        delete pErrorMessage, pErrorMessage = NULL;
 }

 WaitForPrintThread();
 if((m_eCurrentDiscPos == IRoboCommNode::ERLN_RECORDER) ||
                  (m_eCurrentDiscPos == IRoboCommNode::ERLN_PRINTER))
 {
    CN_LOCK;
    CUserDialogHandle *pDialog = m_pCommNode->CreateMoveCDDialog(
                                        m_eCurrentDiscPos,
                                        IRoboCommNode::ERLN_WASTEBIN);
    CN_UNLOCK;
    m_pCommNode->SuspendThread(SIMULATEROBOMOVEMENT_TIMEOUT);

    CN_LOCK;
    if(pDialog != NULL)
        delete pDialog, pDialog = NULL;
    CN_UNLOCK;

    m_eCurrentDiscPos = IRoboCommNode::ERLN_WASTEBIN;
    bResult           = m_bHasNewInterface ?
                              !m_pCommNode->IsAbortedImmediate() : TRUE;
 }

 return bResult;
}
```

5.3.11. StartPrintLabel

This method is called after successfully burning a disc.

```
BOOL CDummyRoboDriver::StartPrintLabel()
{
 if(m_bShowInfoMessage)
 {
    CUserDialogHandle *pErrorMessage = m_pCommNode->CreateMessage(
        "StartPrintLabel is called.\n" \
        "PRINT the LABEL if not already done.\n" \
        "Press Cancel, so these messages will not be displayed
                                anymore.", ROBOCOMMNODE_MSG_HINT);
    if(pErrorMessage != NULL)
        pErrorMessage->WaitForUserFeedback();

    m_bShowInfoMessage = (pErrorMessage->GetFeedback() !=
```

```
                                 CUserDialogHandle::DLGRESULT_CANCEL);

    if(pErrorMessage != NULL)
        delete pErrorMessage, pErrorMessage = NULL;
 }

 return TRUE;
}
```

5.3.12. StartRoboTest

This method is called to test all relevant functionality of the robot.

```
BOOL CDummyRoboDriver::StartRoboTest()
{
 BOOL bResult = TRUE;
 m_bIsTesting = TRUE;

 if(m_bShowInfoMessage)
 {
    CUserDialogHandle *pErrorMessage = m_pCommNode->CreateMessage(
        "StartRoboTest is called.\n" \
        "MOVE the disc USUAL WAY to check the setting.\n" \
        "Press Cancel, so these messages will not be displayed
                                    anymore.", ROBOCOMMNODE_MSG_HINT);
    if(pErrorMessage != NULL)
        pErrorMessage->WaitForUserFeedback();

    m_bShowInfoMessage = (pErrorMessage->GetFeedback() !=
                                 CUserDialogHandle::DLGRESULT_CANCEL);

    if(pErrorMessage != NULL)
        delete pErrorMessage, pErrorMessage = NULL;
 }

 bResult= InsertCD() ? PostBurnOK() : FALSE;

 m_bIsTesting = FALSE;

 return bResult;
}
```

5.3.13. ExitBurn

This method is called after a recording session has been completed.

```
BOOL CDummyRoboDriver::ExitBurn()
{
 if(m_bShowInfoMessage)
 {
     CUserDialogHandle *pErrorMessage = m_pCommNode->CreateMessage(
         "ExitBurn is called.\n" \
         "CLEANUP because burning could be aborted.\n" \
         "Press Cancel, so these messages will not be displayed
                                  anymore.", ROBOCOMMNODE_MSG_HINT);
     if(pErrorMessage != NULL)
         pErrorMessage->WaitForUserFeedback();

     m_bShowInfoMessage = (pErrorMessage->GetFeedback() !=
                              CUserDialogHandle::DLGRESULT_CANCEL);

     if(pErrorMessage != NULL)
         delete pErrorMessage, pErrorMessage = NULL;
 }

 return WaitForPrintThread() ? CleanUp() : FALSE;
}
```

5.3.14. PrintThreadEntry

```
UINT CDummyRoboDriver::PrintThreadEntry(void *pData)
{
 UINT uiResult         = 1;
 CDummyRoboDriver *pThis  = (CDummyRoboDriver*)pData;
 if(pThis != NULL)
     uiResult = pThis->PrintThread();

 return uiResult;
}
```

5.3.15. PrintThread

```
UINT CDummyRoboDriver::PrintThread()
{
 BOOL bResult = FALSE;

 if(m_pCommNode != NULL)
 {
```

Add this thread to our list, so *Nero* will know to wait for it.

```
    if(m_bHasNewInterface)
        m_pCommNode->AddMyPrintThread();

    CUserDialogHandle *pDialog = NULL;

    CN_LOCK;
    pDialog = m_pCommNode->CreatePrintCoverDialog();
    CN_UNLOCK;
    m_pCommNode->SuspendThread(SIMULATEROBOMOVEMENT_TIMEOUT);
    CN_LOCK;
    if(pDialog != NULL)
        delete pDialog, pDialog = NULL;
    CN_UNLOCK;

    if(m_bHasNewInterface ? !m_pCommNode->IsAbortedImmediate() : TRUE)
    {
        CN_LOCK;
        pDialog = m_pCommNode->CreateMoveCDDialog(
                                        IRoboCommNode::ERLN_PRINTER,
                                        IRoboCommNode::ERLN_OUTPUT);

        CN_UNLOCK;
        m_pCommNode->SuspendThread(SIMULATEROBOMOVEMENT_TIMEOUT);
        CN_LOCK;
        if(pDialog != NULL)
            delete pDialog, pDialog = NULL;
        CN_UNLOCK;
    }
    else
        bResult = FALSE;


    if(m_bHasNewInterface)
        m_pCommNode->RemoveMyPrintThread();
 }
```

Return 0 on success.

```
 return bResult ? 0 : 1;
}
```

5.3.16. WaitForPrintThread

Wait for the print thread to exit.

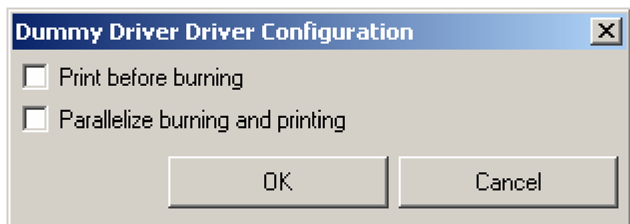
```
BOOL CDummyRoboDriver::WaitForPrintThread()
{
 DWORD dwResult = 1;
 if((m_pPrintThread != NULL) && (m_pCommNode != NULL))
 {
     while(!m_pPrintThread->GetExitCodeThread(&dwResult))
         m_pCommNode->SuspendThread(100);

     delete m_pPrintThread; m_pPrintThread = NULL;
 }
```

If the exit code is 0, everything went fine.

```
 return (dwResult == 0);
};
```

5.4. CDummyRoboDriverConfig Class

This is the configuration class of the dummy robot driver.




```
#include "AbstractUserDialog.h"
#include "DummyDriver.h"

class CDummyRoboDriverConfig :public IUserDialogWatchdog
{
public:
 CDummyRoboDriverConfig(IUserDialogFactory *pDialogFactory);
 virtual ~CDummyRoboDriverConfig();
```

The class is derived from IUserDialogWatchdog to receive every control status change. The method provides the name of the affected control.

```
 virtual void ValueChanged(const char *szName);
 virtual void Exec();

 void OnPrintOrderChanged();
 void OnQuit();
 void OnOk();

private:
 void CreateDialogContent();
```

```
private:
 IUserDialogFactory    *m_pDialogFactory;
 IUserDialog           *m_pDialog;
 IUserDialogElement    *m_pParallelize;
};
```

5.4.1. Constructor

```
CDummyRoboDriverConfig::CDummyRoboDriverConfig(
     IUserDialogFactory *pDialogFactory):m_pDialogFactory(pDialogFactory)
{
 m_pDialog      = NULL;
 m_pParallelize = NULL;
```

Install the watchdog to receive any dialog changes on user interaction.

```
 if(m_pDialogFactory != NULL)
     m_pDialogFactory->InstallWatchdog(this);
}
```

5.4.2. Destructor

No additional operations are performed in the destructor.

```
CDummyRoboDriverConfig::~CDummyRoboDriverConfig()
{
}
```

5.4.3. ValueChanged

Variable changes are notified by calling this method.

```
void CDummyRoboDriverConfig::ValueChanged(const char *szName)
{

#define VCON(name,call) if (!strcmp(name,szName)) call();

 VCON(ABSTRACTUSERDIALOG_CANCEL, OnQuit);
 VCON("OK",                      OnOk);
 VCON(CONFIG_PRINTBEFOREBURN, OnPrintOrderChanged);

#undef VCON
}
```

5.4.4. Exec

This function is called when a configuration dialog is actually executed.

```
void CDummyRoboDriverConfig::Exec()
{
 if(m_pDialogFactory != NULL)
 {
     const char *szDriverName     =
                          m_pDialogFactory->GetOEMName(DUMMYDRIVER_ID);
     const char *szTitleTemplate  = "%s Driver Configuration";
     char *szWindowTitle          = NULL;

     if((szDriverName != NULL) && (szTitleTemplate != NULL))
     {
         szWindowTitle = new char[strlen(szDriverName) +
                                          strlen(szTitleTemplate) + 1];
         if(szWindowTitle != NULL)
             sprintf(szWindowTitle, szTitleTemplate, szDriverName);
     }
```

At first create our configuration dialog window with the driver name in the title.

```
     m_pDialog = m_pDialogFactory->CreateUserDialog(szWindowTitle);
     if(szWindowTitle != NULL)
         delete [] szWindowTitle, szWindowTitle = NULL;

     if(m_pDialog != NULL)
     {
```

Create the two checkboxes.

```
         CreateDialogContent();
```

Create the default 'OK' and 'Cancel' buttons.

```
         IUserDialogContainer *pButtons =
                          m_pDialog->CreateHBox(DE_OPTION_ALIGN_RIGHT);
         if(pButtons != NULL)
         {
             pButtons->CreatePushButton(100, 0, DE_OPTION_ALIGN_CENTER,
                                                          "OK", "OK");
             pButtons->CreatePushButton(100, 0, DE_OPTION_ALIGN_CENTER,
                                ABSTRACTUSERDIALOG_CANCEL, "Cancel");
         }

         m_pDialog->Exec();
         delete m_pDialog; m_pDialog=NULL;
     };
 };
}
```

5.4.5. OnPrintOrderChanged

This function is called when a change of the print order has been detected by the ValueChanged function.

```
void CDummyRoboDriverConfig::OnPrintOrderChanged()
{
 if((m_pParallelize != NULL) && (m_pDialogFactory != NULL))
 {
     BOOL bCanParallelize =!m_pDialogFactory->GetConfigurationValueBOOL(
                                    CONFIG_PRINTBEFOREBURN,
                                    DEFAULTCONFIG_PRINTBEFOREBURN);
     if(bCanParallelize)
         m_pParallelize->SetSensitive(TRUE);
     else
     {
         m_pDialogFactory->SetConfigurationValueBOOL(
                                    CONFIG_PARALLELIZE, FALSE);
         m_pParallelize->SetSensitive(FALSE);
     }
 }
}
```

5.4.6. OnQuit

This function is called when ABSTRACTUSERDIALOG_CANCEL has been detected by the ValueChanged function.

```
void CDummyRoboDriverConfig::OnQuit()
{
 if(m_pDialog != NULL)
     m_pDialog->Quit();
}
```

5.4.7. OnOk

This function is called when a change to the OK control has been detected by the ValueChanged function.

```
void CDummyRoboDriverConfig::OnOk()
{
 if(m_pDialog != NULL)
     m_pDialog->Quit();
}
```

5.4.8. CreateDialogContent

Create two checkboxes for “printing before burning” and parallelizing burning and printing.

```
void CDummyRoboDriverConfig::CreateDialogContent()
{
 m_pDialog->CreateCheckBox(DE_OPTION_ALIGN_LEFT,
                             CONFIG_PRINTBEFOREBURN,
                             "Print before burning",
                             DEFAULTCONFIG_PRINTBEFOREBURN);

 m_pParallelize = m_pDialog->CreateCheckBox(DE_OPTION_ALIGN_LEFT,
                                    CONFIG_PARALLELIZE,
                                    "Parallelize burning and printing",
                                    DEFAULTCONFIG_PARALLELIZE);
 OnPrintOrderChanged();
}
```

5.5. DummyDriverFirmwareCheck.h

This file contains definitions for the firmware check interface.

```
#include "CCOMSerial.h"
#include "NeroRoboFirmwareCheck.h"


#define FW_BUFSIZE        20
#define DUMMY_INVALIW_FW  "InvalidDriver FW ??"

static const char *FirmwareList[] =
{
 "DummyDriver FW 1.11"
};

static const char *PrinterList[] =
{
 "Dummy Robo Printer"
};
```

5.6. CDummyRoboDriverFirmwareCheck

This class offers information on the firmware of the connected robot.

```
class CDummyRoboDriverFirmwareCheck : public CNeroRoboFirmwareCheck
{
public:
 CDummyRoboDriverFirmwareCheck(CSerial *pInterface);
 virtual ~CDummyRoboDriverFirmwareCheck();

 virtual char *GetRoboFirmware();
 virtual bool IsValidFirmware(const char *pcFirmware);
 virtual bool IsValidPrinter(const char *pcPrinterName);

 virtual SerialPortType GetInterfaceType();

private:
 char   *m_pcFirmware;
 CSerial*m_pInterface;
};
```

5.6.1. Constructor

```
CDummyRoboDriverFirmwareCheck::CDummyRoboDriverFirmwareCheck(
                                                   CSerial *pInterface)
{
 m_pcFirmware   = NULL;
```

We accept only serial (COM) ports.

```
 m_pInterface   = ((pInterface != NULL) && (pInterface->GetPortType() == 
PORT_SERIAL)) ? pInterface : NULL;
}
```

5.6.2. Destructor

```
CDummyRoboDriverFirmwareCheck::~CDummyRoboDriverFirmwareCheck()
{
 if(m_pcFirmware != NULL)
     delete [] m_pcFirmware, m_pcFirmware = NULL;
}
```

5.6.3. GetRoboFirmware

This function is called to retrieve the robot's firmware information.

```
char *CDummyRoboDriverFirmwareCheck::GetRoboFirmware()
{
 if(m_pInterface != NULL)
 {
     if(m_pcFirmware != NULL)
         delete [] m_pcFirmware, m_pcFirmware = NULL;

     m_pcFirmware = new char[FW_BUFSIZE];

     if(m_pcFirmware != NULL)
         strcpy(m_pcFirmware, FirmwareList[0]);
 }

 return m_pcFirmware;
}
```

5.6.4. IsValidFirmware

Determine if the current firmware version is valid.

```
bool CDummyRoboDriverFirmwareCheck::IsValidFirmware(const char
*pcFirmware)
{
 return (pcFirmware != NULL) ? !strcmp(FirmwareList[0], pcFirmware) :
false;
}
```

5.6.5. IsValidPrinter

Determine if the attached printer is valid.

```
bool CDummyRoboDriverFirmwareCheck::IsValidPrinter(const char
*pcPrinterName)
{
 return (pcPrinterName != NULL) ? !strcmp(PrinterList[0], pcPrinterName)
: false;
}
```

5.6.6. GetInterfaceType

Retrieve the interface type. Only serial communication is supported.

```
SerialPortType CDummyRoboDriverFirmwareCheck::GetInterfaceType()
{
 return PORT_SERIAL;
}
```

6. Robot Driver Classes and Interfaces

This section describes which classes and interfaces have to be used, if you want to implement your own robot driver.

The robot driver has to be implemented in form of a DLL. This DLL is required to expose some special *Nero* Plug-In functions to be recognized by *Nero* as valid component.

6.1. Naming Conventions

The names of new robot drivers are expected to be prefixed by “NR” and have the extension “.dll”. For example, a Dummy Driver that simulates real robot movements could be named “**NR**DummyDriver.dll”.

If the “NR” prefix or the “.dll” extension is missing, the *Nero* Robot Manager will not load the driver.

6.2. General Nero Plug-in Interface

The following functions have to be implemented.

6.2.1. NERO_PLUGIN_GetVersion

Return the version of the robot driver. The higher 2 Bytes of returned DWORD value define the major and lower 2 Bytes the minor version number. This version will be displayed point separated (e.g. MajorNo.MinorNo).

```
NERO_PLUGIN_API DWORD NERO_PLUGIN_GetVersion();
```

6.2.2. NERO_PLUGIN_GetInterfaceVersion

Return the version of the interface between *Nero* and the driver. Here we should return ROBODRIVER_CURRENTVERSION defined in RoboDriver.h

```
NERO_PLUGIN_API DWORD NERO_PLUGIN_GetInterfaceVersion();
```

6.2.3. NERO_PLUGIN_SetInterfaceVersion

Set the expected interface version. This way a driver newer than the currently used *Nero* version can still support the older version.

```
NERO_PLUGIN_API void  NERO_PLUGIN_SetInterfaceVersion(
            DWORD dwRequestedVersion);
```

6.2.4. NERO_PLUGIN_GetPluginType

Return the type of the plug-in. In case of robot driver we should return ROBODRIVER_PLUGINTYPE defined in RoboDriver.h

```
NERO_PLUGIN_API const char *NERO_PLUGIN_GetPluginType();
```

6.2.5. NERO_PLUGIN_GetPluginName

Return the name of your robot driver (e.g. “Dummy Driver”).

```
NERO_PLUGIN_API const char *NERO_PLUGIN_GetPluginName();
```

6.2.6. NERO_PLUGIN_SetVendorInformations

Gets called from the application, which will provide company and product name. Usually not used by robot driver.

```
NERO_PLUGIN_API void NERO_PLUGIN_SetVendorInformations(
                      const char *szCompany,
                      const char *szProductName);
```

6.3. Robot Driver DLL Interface

Additional to standard *Nero* plug-in interface a robot driver has to implement special export functions used to configure and instantiate the driver or to check the driver and robot properties respectively.
This interface is defined in RoboDriver.h.

6.3.1. nerorobodriver_configuredriver

Obsolete configure driver interface. Older *Nero* versions are still using it.

```
NEROROBODRIVER_API CNeroRoboDriver *nerorobodriver_configuredriver(
                                 IUserDialogFactory *pDialogFactory);
```

6.3.2. nerorobodriver_createdriver

Create a *Nero* robot driver object and connect it to the IRoboCommNode communication node that was passed as parameter.

The node is only borrowed by the robot driver and despite the convention that it is reserved for the exclusive use through the robot driver, it is still owned by the caller of this DLL entry point and may not be deleted.

Those methods are obsolete and replaced by methods below, but for compatibility to older *Nero* versions the driver should implement these methods too.

```
NEROROBODRIVER_API CNeroRoboDriver *nerorobodriver_createdriver(
                                 IRoboCommNode *pCommNode);
```

6.3.3. nerorobodriverV2_createdriver

If we have the possibility to assign the recorder location to its unique name, we can tell the robot driver which recorder it's serving. If there are not multiple recorders on the system with the same name, we pass a valid, unique recorder name to this method. Otherwise, the recorder name string will be **empty**.

```
NEROROBODRIVER_API CNeroRoboDriver *nerorobodriverV2_createdriver(
                                 IRoboCommNode *pCommNode,
                                 const char *cpRecorderName);
```

Identifier	Introduced in *Nero* version
nerorobodriverV2_createdriver	5.5.9.8

6.3.4. nerorobodriverV2_configuredriver

The driver can configure the robot depending on the passed unique recorder name it should serve.

The recorder name string is valid only for unique recorder names.

```
NEROROBODRIVER_API void nerorobodriverV2_configuredriver(
                                 IUserDialogFactory *pDialogFactory,
                                 const char *cpRecorderName);
```

Identifier	Introduced in *Nero* version
nerorobodriverV2_configuredriver	5.5.9.8

6.3.5. nerorobodriver_firmwarecheck

Check the current robot firmware. This information is used by *Nero* for automatic robot recognition.

```
NEROROBODRIVER_API CNeroRoboFirmwareCheck *nerorobodriver_firmwarecheck(
                                 CSerial *pInterface);
```

6.4. Robot Driver Configuration Interface

IUserDialogWatchdog

Implement this interface if you want to stay up to date about your configuration information and also to make your dialog dynamic.

IConfigurationStorage

NOTE: The current *Nero* implementation of IUserDialogFactory, which derives from IConfigurationStorage to save configuration data in the registry, uses a unique robot driver ID as key. This key consists of the port type ID the robot is connected to, port number, SCSI adapter host ID of recorder the driver is assigned to, SCSI target ID of the same recorder and the robot name. This applies for the case where multiple recorders with the same name exist in the system.

If we have only one recorder or multiple, different recorder names the implementation of IConfigurationStorage in *Nero* will use this recorder name to create registry key for configuration storage.

6.4.1. IConfigurationStorage::InstallWatchdog

Register watchdog handler, which should receive value change notifications.

```
virtual void InstallWatchdog(IUserDialogWatchdog *pDialogWatchdog)
                                                                = 0;
```

6.4.2. IConfigurationStorage::UninstallWatchdog

This method should be called if value change notifications should not be received anymore.

```
virtual void UninstallWatchdog(IUserDialogWatchdog
                                              *pDialogWatchdog) = 0;
```

6.4.3. IConfigurationStorage::SendNotification

This function will be called by the user interface if some named values changed, to inform all watch dogs.

```
virtual void SendNotification(const char *szName) const
                                                                = 0;
```

6.4.4. IConfigurationStorage::GetConfigurationValue

Get string configuration value type. If there is not appropriate value already set, the default value will be returned.

```
virtual const char *GetConfigurationValue(const char *szName,
                                        const char *szDefault)   = 0;
```

6.4.5. IConfigurationStorage::GetConfigurationValueBOOL

Get Boolean configuration value type. If there is not appropriate value already set, the default value will be returned.

```
virtual BOOL GetConfigurationValueBOOL(const char *szName,
                                        BOOL bDefault)          = 0;
```

6.4.6. IConfigurationStorage::GetConfigurationValueINT

Get integer configuration value type. If there is not appropriate value already set, the default value will be returned.

```
virtual int  GetConfigurationValueINT(const char *szName,
                                        int iDefault )          = 0;
```

6.4.7. IConfigurationStorage::SetConfigurationValue

Set string configuration value type.

```
virtual void SetConfigurationValue(const char *szName,
                                 const char *szValue)           = 0;
```

6.4.8. IConfigurationStorage::SetConfigurationValueBOOL

Set Boolean configuration value type.

```
virtual void SetConfigurationValueBOOL(const char *szName,
                                        BOOL bValue)                = 0;
```

6.4.9. IConfigurationStorage::SetConfigurationValueINT

Set integer configuration value type.

```
virtual void SetConfigurationValueINT(const char *szName,
                                       int iValue)                  = 0;
```

6.5. Robot Driver Firmware Interface

The CNeroRoboFirmwareCheck class defines an additional robot driver interface, which enables *Nero* to determine the firmware of connected robots. This interface is used by *Nero* mainly to automatically recognize robots if a particular port was selected in the configuration dialog.

6.5.1. GetRoboFirmware

Read firmware version from named robot.

```
virtual char *GetRoboFirmware()
```

Identifier	Introduced in *Nero* version
GetRoboFirmware	5.5.9.8

6.5.2. IsValidFirmware

Check if the received firmware belongs to a known robot. Here the firmware string received in method above will be passed, so the driver can check if it is a valid string. If this method returns true, *Nero* will automatically assign this robot driver to the recorder, which was selected in the configuration dialog.

```
virtual bool IsValidFirmware(const char *cpFirmware);
```

Identifier	Introduced in *Nero* version
IsValidFirmware	5.5.9.8

6.5.3. IsValidPrinter

Check also for valid printer name we found on local machine. There could be a couple of special robot printers which also can be automatically assigned to selected robot driver.

```
virtual bool IsValidPrinter(const char *cpPrinterName);
```

Identifier	Introduced in *Nero* version
IsValidPrinter	5.5.9.8

6.5.4. GetInterfaceType

Save the supported interface types in the driver. This method should return the port type the robot is using for communication.

```
virtual SerialPortType GetInterfaceType();
```

Identifier	Introduced in *Nero* version
GetInterfaceType	5.5.10.2

6.6. Robot Driver Move Interface

The following methods define entry points for the different hook-ups within the *Nero* Burn process. A *NeroROBO* driver will implement whichever methods it may need to perform its tasks. All other methods may remain unimplemented with the exception of InsertCD() and RemoveNonWriteableCD().

All those methods shall return a Boolean TRUE if all desired actions could be performed successfully and FALSE otherwise.

Note that the recording process will stop if FALSE is returned and PostBurn()/ExitBurn() will be called unless the error occurred in one of those methods.

6.6.1. InitBurn

The InitBurn method is called just after loading the robot driver DLL.

Usually, a driver will want to perform its reset function or similar jobs at this point.

Note that this method is only called once during the whole recording job, even if more than one copy is requested.

```
virtual BOOL InitBurn();
```

Identifier	Introduced in *Nero* version
InitBurn	5.5.4.0

6.6.2. PreBurn

This method is called for every recording cycle right before it will start. This is the place where your driver should call the PrintLabel() method for those cases where the label needs to be printed prior to burning.

```
virtual BOOL PreBurn();
```

Identifier	Introduced in *Nero* version
PreBurn	5.5.4.0

6.6.3. InsertCD

This method is called if *Nero* needs a disc to write to.

The driver should ensure that there is a disc in the tray of the recorder when this method returns. Otherwise, it must return FALSE.

Note that the state of the recorder tray at the time this method is called is undefined. Therefore, prior to actually trying to insert a disc, the driver will have to explicitly open the tray with IRoboCommNode::MoveRecorderTray(TRUE).

The essential point about all this is that this way the robot driver can decide when the tray will be opened which is very useful if some job needs to be done with a medium prior to inserting it into the drive (e.g. printing).

```
virtual BOOL InsertCD();
```

Identifier	Introduced in *Nero* version
InsertCD	5.5.4.0

6.6.4. RemoveNonWriteableCD

This method will be called if *Nero* encounters a disc it cannot write to.

It is the responsibility of the robot driver to remove the inapt media in this case.

When this method is called, the tray is open.

```
virtual BOOL RemoveNonWriteabaleCD();
```

Identifier	Introduced in *Nero* version
RemoveNonWriteableCD	5.5.4.0

6.6.5. RemoveUntouchedCD

This method will be called if *Nero* decides not to use a disc after all but it has already been inserted into the drive. This gives the robot a chance to put the disc back into the input tray or wherever it may decide to put it (may as well be the waste bin of course).

```
virtual BOOL RemoveUntouchedCD();
```

Identifier	Introduced in *Nero* version
RemoveUntouchedCD	5.5.4.0

6.6.6. PostBurnOK

This function is called when *Nero* is done burning a disc.

PostBurnOK will be called if the recording process was successful.

The driver should take this into account when deciding where to put the resulting disc (e.g. if the recording process was not successful, the driver might want to put the disc into a trash bin and into some output bin otherwise).

The recorder's tray is open when this method is called.

```
virtual BOOL PostBurnOK();
```

Identifier	Introduced in *Nero* version
PostBurnOK	5.5.4.0

6.6.7. PostBurnFAILURE

This function is called when *Nero* is done burning a disc.

PostBurnFAILURE will be called if the recording process failed for some reason.

The driver should take this into account when deciding where to put the resulting disc (e.g. if the recording process was not successful, the driver might want to put the disc into a trash bin and into some output bin otherwise).

The recorder's tray is open when this method is called.

```
virtual BOOL PostBurnFAILURE();
```

Identifier	Introduced in *Nero* version
PostBurnFAILURE	5.5.4.0

6.6.8. ExitBurn

This method is called to conclude the recording process.

It is called only once for a recording job, even if this job created more than one copy.

This hook can be used to clean up any structures specific to the current recording process. The recorder tray is closed when this method is called.

```
virtual BOOL ExitBurn();
```

Identifier	Introduced in *Nero* version
ExitBurn	5.5.4.0

6.6.9. CleanUp

Perform any cleanup jobs that may be necessary after the robot has been left in an undefined state (remove any remaining media from the printer tray etc.)

```
virtual BOOL CleanUp();
```

Identifier	Introduced in *Nero* version
CleanUp	5.5.4.0

6.6.10. StartPrintLabel

Explicitly tell the robot to start printing a label. In this way we are able to print with delay.

This method will be explicit called after each successful burning. The driver can ignore this call if printing was already done in the PostBurnOK method or the label was printed before burning. The difference is, if we call this method InsertCD was already called for a new copy, so there should be disc in printer and recorder.

```
virtual BOOL StartPrintLabel();
```

Identifier	Introduced in *Nero* version
StartPrintLabel	5.5.9.4

6.6.11. StartRoboTest

A simple robot test. The driver may move the disc through all robot positions (e.g. sequence: input bin, recorder, printer, output bin) to test proper configuration.

```
virtual BOOL StartRoboTest();
```

Identifier	Introduced in *Nero* version
StartRoboTest	5.5.9.4

6.7. Serial Port Interface

CSerial is a compatibility class for multiplatform serial port access.

Note that 'serial port' refers to all interfaces like USB, COM, and LPT receiving data sequentially.

This class is pure virtual.

6.7.1. SerialPortType Enum

The following enumeration defines different port types supported by *Nero*.

Do not change the order of these constants as their values may be stored in configuration profiles.

Note: Currently (*Nero 6*) only serial port (COM) support is implemented by *Nero*.

```
typedef enum
{
 PORT_SERIAL       = 0,
 PORT_PARALLEL     = 1,
 PORT_USB          = 2,
 PORT_BRAVO        = 3
} SerialPortType;
```

6.7.2. CSerial::OpenPort

This function attempts to open the configured port. It returns TRUE on success and FALSE on failure.

```
 virtual BOOL OpenPort()                    = 0;
```

6.7.3. CSerial::ReadData

Returns number of bytes read/written or -1 in case of an error.

Will return 0 if no bytes could be read until the timeout was reached.

```
virtual int ReadData(char *buffer, int iBytes,
                     int timeout = CSERIALTIMEOUT_NONE)      = 0;
```

Description of parameters	
buffer	Buffer being read from.
iBytes	Number of bytes read.
timeout	Maximum time the operation should take in ms.

6.7.4. CSerial::WriteData

Returns number of bytes read/written or -1 in case of an error.

```
virtual int WriteData(const char *buffer, int iBytes,
                     int timeout=CSERIALTIMEOUT_NONE)        = 0;
```

Description of parameters	
buffer	Buffer being written into.
iBytes	Number of bytes written.
timeout	Maximum time the operation should take in ms.

6.7.5. CSerial::FlushInput

Flush input buffer. Warning! All characters in the buffer will be lost forever. However, this is necessary to get the lines into a defined state.

```
virtual void FlushInput()                                       = 0;
```

6.7.6. CSerial::FlushOutput

Flush output buffer. Warning! All characters in the buffer will be lost forever. However, this is necessary to get the lines into a defined state, e.g. before sending a command.

```
virtual void FlushOutput()                                      = 0;
```

6.7.7. CSerial::ClosePort

Close the port. Returns TRUE for success and FALSE for error.

```
virtual BOOL ClosePort()                                        = 0;
```

6.7.8. CSerial::GetPortType

Return type of serial port.

```
virtual SerialPortType GetPortType()                                = 0;
```

6.7.9. CSerial::GetPortNum

Return the port number of this port on the current system.

```
virtual int GetPortNum()                                            = 0;
```

6.7.10. CSerial::CreateInterface

Create a serial connection of specified type.

The port number starts counting from 1 (e.g. COM1, COM2... LPT1, LPT2...).

```
static CSerial *CreateInterface(SerialPortType type, int iPortNum,
                                    const char *cpPrinterName = NULL);
```

6.7.11. CSerial::CreateCOMPortInterface

The following methods will create specific types of serial connections.

They are invoked by the CreateInterface method.

Note that in contrast to above method, this method will always return a new serial port object and not a sharing proxy.

```
static CSerial *CreateCOMPortInterface(int iPortNum,int iBaudRate);
```

6.7.12. CSerial::CreateUSBPortInterface

Dummy implementation for USB. Not implemented yet.

```
static CSerial *CreateUSBPortInterface(int iPortNum,int iBaudRate);
```

6.7.13. CSerial::CreateBravoInterface

Support for the Bravo robot.

```
static CSerial *CreateBravoInterface(const char *cpPrinterName);
```

7. User Dialog Interface

This section deals with the different abstract classes that represent the user dialog interface.

7.1. CUserDialogHandle

7.1.1. CUserDialogHandle::UserDlgFeedbackType

User feedback enumeration. DLGRESULT_NONE means no user feedback received.

```
typedef enum
 {
    DLGRESULT_OK,
    DLGRESULT_CANCEL,
    DLGRESULT_NONE
 }
 UserDlgFeedbackType;
```

7.1.2. CUserDialogHandle::WaitForUserFeedback

Waits for the user to answer the dialog.

```
 virtual void WaitForUserFeedback()          = 0;
```

7.1.3. CUserDialogHandle::GetFeedback

Retrieve feedback. Will return the type of button the user has pressed to quit the dialog.

Querying this method should be enough for most dialogs. However, if additional information is obtained from the user, GetFeedbackData can be used to return this information.

```
 virtual UserDlgFeedbackType GetFeedback()   = 0;
```

7.1.4. CUserDialogHandle::GetFeedbackData

Get the result of the dialog. Depending on the kind of dialog this handle is associated with, this may e.g. return a string or a filename.

The default implementation of this method will just return NULL.

```
 virtual const char *GetFeedbackData()       { return NULL; };
```

7.1.5. CUserDialogHandle::~CUserDialogHandle

Close the dialog and destroy its handle.

```
 virtual ~CUserDialogHandle()            {};
```

7.2. IUserDialogElement

Interface enabling limited manipulation of dialog elements.

7.2.1. IUserDialogElement::SetShow

Show (bShow = TRUE) and hide (bShow = FALSE) the current dialog element respectively.

```
virtual void SetShow(BOOL bShow)                     = 0;
```

7.2.2. IUserDialogElement::SetSensitive

Enable (bSensitive = TRUE) and disable (bSensitive = FALSE) the current dialog element respectively.

```
virtual void SetSensitive(BOOL bSensitive)           = 0;
```

7.2.3. IUserDialogElement::SetPosition

Called by the container to set the position of dialog element.

```
virtual void SetPosition(int xpos, int ypos)             = 0;
```

7.2.4. IUserDialogElement::GetSize

Retrieve the size of dialog element.

```
virtual void GetSize(int &iXSize, int &iYSize) const  = 0;
```

7.3. IUserDialogContainer

Options valid for every container element:

```
#define DE_OPTION_ALIGN_LEFT   0
#define DE_OPTION_ALIGN_RIGHT  1
#define DE_OPTION_ALIGN_CENTER 3
```

7.3.1. IUserDialogContainer::CreateCheckBox

Factory methods for checkbox control.

```
virtual IUserDialogElement   *CreateCheckBox(DWORD dwOptions,
                                const char *szName,
                                const char *szText,
                                BOOL bDefaultState = FALSE)  = 0;
```

7.3.2. IUserDialogContainer::CreatePushButton

Factory method for pushbutton control.

```
virtual IUserDialogElement   *CreatePushButton(int iMinXsize,
                                int iMinYsize,
                                DWORD dwOptions,
                                const char *szName,
                                const char *szText)          = 0;
```

7.3.3. IUserDialogContainer::CreateHBox

Factory method for textbox control with horizontal scrollbar.

```
virtual IUserDialogContainer *CreateHBox(DWORD dwOptions = 0)    = 0;
```

7.3.4. IUserDialogContainer::CreateVBox

Factory method for textbox control with vertical scrollbar.

```
virtual IUserDialogContainer *CreateVBox(DWORD dwOptions = 0)    = 0;
```

7.3.5. IUserDialogContainer::SizeChanged

Called by a child to notify the parent about a size change.

```
virtual void SizeChanged(IUserDialogElement *pElement)           = 0;
```

7.3.6. IUserDialogContainer::Unref

Called by a child's destructor.

```
virtual void Unref(IUserDialogElement *pElement)                 = 0;
```

7.4. IUserDialog

7.4.1. IUserDialog::Quit

Call this from within a watchdog callback to quit the dialog.

```
virtual void Quit()      = 0;
```

7.4.2. IUserDialog::Exec

Call this to execute the dialog.

```
virtual void Exec()      = 0;
```

7.5. IUserDialogFactory

7.5.1. IUserDialogFactory::CreateUserDialog

Create a dialog. The dialog is being shown immediately with szTitle and align options specified in IUserDialogContainer. Default all dialog elements are left aligned.

```
virtual IUserDialog *CreateUserDialog(const char *szTitle,
                                      DWORD dwOptions = 0)     = 0;
```

7.5.2. IUserDialogFactory::GetCurrentLanguage

This method can be used to determine the language currently used by *Nero* to localize driver messages.

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

7.5.3. IUserDialogFactory::GetOEMName

Return the currently set OEM name for the device given in szTemplateName.

If no matching OEM substitution could be found, return szTemplateName.

```
virtual const char *GetOEMName(const char *szTemplateName)        =
0;
```

7.5.4. IUserDialogFactory::CreateMessageBox

Create simple message box which only allows the user to accept the message.

```
virtual CUserDialogHandle::UserDlgFeedbackType CreateMessageBox(
                                          const char *szText)   = 0;
```

8. Header Files

8.1. RoboDriver.h

```
#include "GeneralPluginDLL.h"
```

Compatibility mode interfaces.

We try to keep backwards compatibility. So this is the layout of the old Version 1 Robot Interface available since *Nero* 5.5.4.0.

CNeroRoboDriverV1

```
class CNeroRoboDriverV1
{
public:
 virtual BOOL InitBurn()             { return TRUE; };
 virtual BOOL PreBurn()              { return TRUE; };
 virtual BOOL InsertCD()             = 0;
 virtual BOOL RemoveNonWriteableCD() = 0;
 virtual BOOL RemoveUntouchedCD()    = 0;
 virtual BOOL PostBurnOK()           { return TRUE; };
 virtual BOOL PostBurnFAILURE()      { return TRUE; };
 virtual BOOL ExitBurn()             { return TRUE; };
 virtual ~CNeroRoboDriverV1()        {};
};
```

Wait 60 seconds for the drive to get ready in DlgWaitCD.

If the drive hasn't managed to read the disc during that time, ask the user to insert a CD or abort instead.

```
#define ROBODRIVER_INSERTDISCTIMEOUT 60000
```

Current driver version used for compatibility. Change this version only if the robot driver interface has changed. Therefore a proxy has to be adapted to this new version number.

```
#define ROBODRIVER_CURRENTVERSION       5
#define ROBODRIVER_PLUGINTYPE           "NeroCDRobo"
#define NEROROBODRIVER_API NERO_PLUGIN_API
```

Forward declarations.

```
class IRoboCommNode;
class IUserDialogFactory;
class CNeroRoboFirmwareCheck;
class CSerial;
```

CNeroRoboDriverV2

Since *Nero* 5.5.4.1, CNeroRoboDriverV1 extends the CNeroRoboDriverV1 interface.

```
class CNeroRoboDriverV2:public CNeroRoboDriverV1
{
public:
 virtual BOOL InitBurn()              { return TRUE; };
 virtual BOOL PreBurn()               { return TRUE; };
 virtual BOOL InsertCD()              = 0;
 virtual BOOL RemoveNonWriteableCD() = 0;
 virtual BOOL RemoveUntouchedCD()    = 0;
 virtual BOOL PostBurnOK()            { return TRUE; };
 virtual BOOL PostBurnFAILURE()       { return TRUE; };
 virtual BOOL ExitBurn()              { return TRUE; };
 virtual BOOL CleanUp()               { return TRUE; };

 virtual ~CNeroRoboDriverV2()     {};
};
```

CNeroRoboDriver

Since *Nero* 5.5.9.4 this is the published interface.

```
class CNeroRoboDriver:public CNeroRoboDriverV2
{
public:
 virtual BOOL StartPrintLabel()   { return TRUE; };
 virtual BOOL StartRoboTest()     { return TRUE; };

 virtual ~CNeroRoboDriver()       {};
};
```

8.2. RoboCommNode.h

This file describes the robot communication node interface that is passed to each robot driver upon initialization.

A robot driver can use this communication node to communicate with its hardware, the user and, in some limited ways, the operating system.

```
#include "AbstractConfigurationStorage.h"

#define TEMPSTORAGEMUTEXNAME "TempStorageAccess"
```

IRoboThread

Abstract thread handle for multithreaded robot drivers.

```
class IRoboThread
{
public:
```

Thread entry point.

```
 typedef UINT(*StartUp)(void *pUserData);
```

If this method returns TRUE, dwResult will contain the exit code of the thread.

If FALSE is returned, the specified thread is still running.

```
 virtual BOOL GetExitCodeThread(DWORD *dwResult) = 0;
 virtual ~IRoboThread() {};
};
```

IRoboMutex

Abstract mutex interface for multithreaded robot drivers.

```
class IRoboMutex
{
public:
```

Lock a mutex, pass -1 as a timeout and it will try to lock the mutex indefinitely.

```
 virtual BOOL Lock(DWORD dwTimeOut = (DWORD)-1) = 0;
 virtual BOOL Unlock()                         = 0;
 virtual ~IRoboMutex() {};
};

class CUserDialogHandle;
class IConfigurationStorage;
class CSerial;
class IRoboPrinter;
```

IRoboCommNodeV1

Since *Nero* 5.5.4.0:

First common node interface for robots.

```
class IRoboCommNodeV1:virtual public IConfigurationStorage
{
public:
```

This function will return a serial interface to the hardware this driver will handle. The caller must **not** free this object with delete.

Nero knows about the relations between drivers and hardware so this method will return an instance of the type of interface *Nero* chose to present to the driver. Also note that multiple calls to this method will always return the same instance of the interface.

```
virtual CSerial *GetSerialInterface()         = 0;
```

Open/Close the recorder's tray. If TRUE is passed to this method, the tray will be opened and closed otherwise.

The method will return TRUE if successful and FALSE in case of an error.

```
virtual BOOL MoveRecorderTray(BOOL bOpen)     = 0;
```

MultiThreading and MultiTasking control

Create a new thread.

```
virtual IRoboThread *CreateThread(
       IRoboThread::StartUp, void* pUserData)  = 0;
```

Create a mutex.

```
virtual IRoboMutex *CreateMutex()             = 0;
```

Suspend the robot thread for a specified amount of time.

```
virtual void SuspendThread(DWORD dwTimeInMS)   = 0;
```

Return a value with only one guaranteed property, namely that the value is increased by 1 every millisecond. Good for timeouts.

```
virtual DWORD GetSyncTime()                   = 0;
```

Returns the currently selected *Nero* Language or NULL if no internationalization support is present.

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

Return the currently set OEM name for a device given in szTemplateName.

If no matching OEM substitution could be found, return szTemplateName.

```
virtual const char *GetOEMName(const char *szTemplateName)   = 0;
```

PrintResult

Printing can either be successful, fail or be aborted.

```
typedef enum
{
    PRINT_OK,
    PRINT_FAILURE,
    PRINT_ABORTED
} PrintResult;
```

This method can be used to determine whether a label should be printed at all.

It will return TRUE if a label should be printed to the CD, i.e. the user specified a printer and label to print.

```
virtual BOOL LabelPrintEnabled()              = 0
```

The driver must call this method at the appropriate time if it wants to print onto the disc. It must ensure that the disc is already in the printer when this method is being called.

```
virtual PrintResult PrintLabel()              = 0;
```

User Interface methods

The following methods will return a user interface handle. It can be used to query the user for actions or to just display some sort of status information while the robot is performing its tasks. The primary thread will take care of keeping the message queue of the GUI running so the robot task can modally wait for a dialog to be closed again or just leave it alone entirely.

Simply delete CUserDialogHandles when done.

ERoboLocationNode Enum

Create a dialog representing the current action the robot device is performing.

```
typedef enum
{
    ERLN_INPUT,
    ERLN_RECORDER,
    ERLN_OUTPUT,
    ERLN_PRINTER,
    ERLN_WASTEBIN
} ERoboLocationNode;

virtual CUserDialogHandle *CreateMoveCDDialog(
                                    ERoboLocationNode source,
                                    ERoboLocationNode destination)   = 0;
```

```
virtual CUserDialogHandle *CreatePrintCoverDialog()                 = 0;
```

Layout of the dialog type mask:

Bits 0-7: Message Type

Bits 8-31: Options

```
#define ROBOCOMMNODE_MSG_ERROR       0
#define ROBOCOMMNODE_MSG_WARNING     1
#define ROBOCOMMNODE_MSG_QUESTION    2
#define ROBOCOMMNODE_MSG_HINT        3
```

Dialogs created this way will return

CUserDialogHandle::DLGRESULT_OK or

CUserDialogHandle::DLGRESULT_CANCEL

```
virtual CUserDialogHandle *CreateMessage(const char *message,
                                                    DWORD options) = 0;
};
```

IRoboCommNodeV2

Since *Nero* 5.5.7.9.

Extended the previous interface for synchronization of multiple robots and especially of robots with multiple drives.

```
class IRoboCommNodeV2:public IRoboCommNodeV1
{
public:
```

Synchronized temporary storage is used to store values accessible to all active robo drivers.

This means that if you know the key to a value, you can synchronize two drivers using this facility.

```
virtual IConfigurationStorage *GetSyncedTemporaryStorage()    = 0;
```

This method can be used to get a global mutex common to all robot drivers.

Use this to synchronize access to temporary storage.

```
virtual void LockNamedMutex(const char *szMutexName)          = 0;
virtual void UnlockNamedMutex(const char *szMutexName)        = 0;
};
```

This Enum can be used to specify the recorder location in method SetRecorderLocation of IRoboCommNode interface below. At this time is only possible to specify left and right recorder. The default recorder is used e.g. for robots with only one drive. Later we have to adapt these values for other robots.

```
typedef enum
{
 DEFAULT_RECORDER  = 0,
 LEFT_RECORDER     = 1,
 RIGHT_RECORDER    = 2
}
ERoboRecorderLocation;
```

IRoboCommNode

Since *Nero* 5.5.9.4.

New CommNode Interface for additional functionality.

```
class IRoboCommNode:public IRoboCommNodeV2
{
public:
```

The driver can use this method to check if the user wishes to immediately abort the current robot movement.

```
 virtual BOOL IsAbortedImmediate()                          = 0;
```

This method is used to move the tray of recorder which is marked as LEFT_RECORDER.

It is specific to robots with two recorders to e.g. close the recorder tray which is located above the waste bin.

```
 virtual void MoveSecondaryRecTray(BOOL bOpen)              = 0;
```

Call this method to inform *Nero* that a print thread was created. This will prevent *Nero* from terminating the burn loop before the label was printed.

```
 virtual void AddMyPrintThread()                            = 0;
```

Remove your print thread after printing was finished.

NOTE: It is a sign of good style to create a separate printing thread. The robot can then perform other operations while the label is created.

```
 virtual void RemoveMyPrintThread()                         = 0;
```

This method specifies the recorder location. The driver can call this method to inform the robot manager if its recorder is the left recorder which will be opened in MoveSecondaryRecTray method.

```
 virtual void SetRecorderLocation(int iRecorderLocation)   = 0;
```

Get current recorder location currently served by the driver.

```
 virtual int GetRecorderLocation()                          = 0;
```

This method enables the robot driver to write an error or status messages to the common NeroHistory.log file. These messages will be appended at the end of the current burn process info.

```
 virtual void WriteErrorToLog(const char *cpErroMsg)       = 0;
};
```

Defines an interface to the printer. Processed by the AbstractRoboGuiManager when the Robot wants to print something.

```
class IRoboPrinter
{
 public:
```

Print and return error. iNumCopy is a counter starting at 1, increased by one with every print job (counter is global, it affects all connected robots).

```
   virtual IRoboCommNode::PrintResult operator ()(int iNumCopy,
                                      IRoboCommNode *pCommNode)= 0;
```

This method returns TRUE if printing is currently enabled.

Another option to disable printing still is to not provide a IRoboPrinter instance at all but this method allows us to be more flexible regarding printer control.

```
   virtual BOOL    IsPrintingEnabled() const               = 0;
```

Create a copy of this exact interface.

```
   virtual IRoboPrinter *Clone() const                    = 0;

   virtual ~IRoboPrinter() {};
};
```

8.3. NeroRoboFirmwareCheck.h

This file contains the declaration of CNeroRoboFirmwareCheck. This class defines an additional robot driver interface, which enables *Nero* to determine the firmware of connected robots. This interface is used by *Nero* mainly to automatically recognize robots if a particular port was selected in the configuration dialog.

```
#include "CSerial.h"
```

CNeroRoboFirmwareCheckV1

Since *Nero* 5.5.9.8.

```
class CNeroRoboFirmwareCheckV1
{
public:
 virtual char *GetRoboFirmware()                          = 0;
 virtual bool IsValidFirmware(const char *cpFirmware)     = 0;
 virtual bool IsValidPrinter(const char *cpPrinterName)   = 0;

 virtual ~CNeroRoboFirmwareCheckV1() {};
};
```

CNeroRoboFirmwareCheck

Since *Nero* 5.5.10.2.

```
class CNeroRoboFirmwareCheck:public CNeroRoboFirmwareCheckV1
{
public:
 virtual SerialPortType GetInterfaceType()  = 0;
 virtual ~CNeroRoboFirmwareCheck()          {};
};
```

8.4. CSerial.h

CSerial is a compatibility class for multiplatform serial port access. Note that 'serial port' refers to all interfaces like USB, COM, LPT receiving data sequentially.

This class is pure virtual.

```
typedef enum
{
 PORT_SERIAL       = 0,
 PORT_PARALLEL     = 1,
 PORT_USB          = 2,
 PORT_BRAVO        = 3
} SerialPortType;


#define CSERIALTIMEOUT_NONE      0
#define CSERIALTIMEOUT_INFINITE -1

class CSerial
{
public:
 virtual ~CSerial() {};

 virtual BOOL OpenPort()                    = 0;
 virtual int ReadData(char *buffer, int iBytes,
                         int timeout = CSERIALTIMEOUT_NONE)      = 0;
 virtual int WriteData(const char *buffer, int iBytes,
                         int timeout=CSERIALTIMEOUT_NONE)        = 0;
 virtual void FlushInput()                                       = 0;
 virtual void FlushOutput()                                      = 0;
 virtual BOOL ClosePort()                                        = 0;
 virtual SerialPortType GetPortType()                            = 0;
 virtual int GetPortNum()                                        = 0;
 static CSerial *CreateInterface(SerialPortType type, int iPortNum,
                                       const char *cpPrinterName = NULL);
 static CSerial *CreateCOMPortInterface(int iPortNum,int iBaudRate);
 static CSerial *CreateUSBPortInterface(int iPortNum,int iBaudRate);

 static CSerial *CreateBravoInterface(const char *cpPrinterName);
```

8.5. CCOMSerial.h

Serial interface for PC COM Ports.

This is an extension to the CSerial interface to allow some COM Port specific configuration for interfaces returning the type PORT_SERIAL at runtime.

```
#include "CSerial.h"
```

CCOMSerial

```
class CCOMSerial
 :public CSerial
{
public:
 typedef enum
 {
    CPARITY_EVEN,
    CPARITY_MARK,
    CPARITY_NO,
    CPARITY_ODD,
    CPARITY_SPACE
 } ParitySettings;

 typedef enum
 {
    STOPBITS_ONE,
    STOPBITS_ONEP5,
    STOPBITS_TWO
 } StopBitSettings;
```

Adjust the configuration of the serial interface. SetConfiguration returns TRUE on success.

Warning: This function must not be called when port is open.

```
 virtual BOOL SetConfiguration(int iBaudRate, int iDataBits,
               ParitySettings psParity, StopBitSettings sbStopBits) = 0;
```

Description of parameters	
iBaudRate	The baud rate at which the communication device operates.
iDataBits	Specifies the number of bits in the bytes transmitted and received.
psParity	Specifies the parity scheme to be used. It is one values above.
sbStopBits	Specifies the number of stop bits to be used. It is one values above.

```
 virtual SerialPortType GetPortType() { return PORT_SERIAL; };
};
```

8.6. AbstractUserDialog.h

This file defines an abstract way to represent a dialog with an interface exported to e.g. a DLL or a separate thread.

```
#include "AbstractConfigurationStorage.h"
```

Event received through the watchdog when dialog is closed.

```
#define ABSTRACTUSERDIALOG_CANCEL "CANCEL"

class CUserDialogHandle
{
public:
 typedef enum
 {
    DLGRESULT_OK,
    DLGRESULT_CANCEL,
    DLGRESULT_NONE
 }
 UserDlgFeedbackType;
 virtual void WaitForUserFeedback()         = 0;
 virtual UserDlgFeedbackType GetFeedback()  = 0;
 virtual const char *GetFeedbackData()      { return NULL; };
 virtual ~CUserDialogHandle()           {};
};

class IUserDialogContainer;

class IUserDialogElement
{
public:
 virtual void SetShow(BOOL bShow)                  = 0;
 virtual void SetSensitive(BOOL bSensitive)        = 0;

 virtual ~IUserDialogElement() {};
 virtual void SetPosition(int xpos, int ypos)          = 0;
 virtual void GetSize(int &iXSize, int &iYSize) const  = 0;
};
```

IUserDialogContainer

```
class IUserDialogContainer:public IUserDialogElement
{
public:
#define DE_OPTION_ALIGN_LEFT   0
#define DE_OPTION_ALIGN_RIGHT  1
#define DE_OPTION_ALIGN_CENTER 3
 virtual IUserDialogElement   *CreateCheckBox(DWORD dwOptions,
```

```
                                    const char *szName,
                                    const char *szText,
                                    BOOL bDefaultState = FALSE)      = 0;

 virtual IUserDialogElement   *CreatePushButton(int iMinXsize,
                                    int iMinYsize,
                                    DWORD dwOptions,
                                    const char *szName,
                                    const char *szText)              = 0;

 virtual IUserDialogContainer *CreateHBox(DWORD dwOptions = 0) = 0;
 virtual IUserDialogContainer *CreateVBox(DWORD dwOptions = 0) = 0;

 virtual ~IUserDialogContainer() {};
 virtual void SizeChanged(IUserDialogElement *pElement)       = 0;
 virtual void Unref(IUserDialogElement *pElement)         = 0;
};
```

IUserDialog

```
class IUserDialog:public IUserDialogContainer
{
public:
 virtual void Quit()      = 0;
 virtual void Exec()      = 0;

 virtual ~IUserDialog()   {};
};
```

IUserDialogFactory

```
class IUserDialogFactory:virtual public IConfigurationStorage
{
public:
 virtual IUserDialog *CreateUserDialog(const char *szTitle,
                                       DWORD dwOptions = 0)      = 0;

 virtual const char *GetCurrentLanguage()                        = 0;
 virtual const char *GetOEMName(const char *szTemplateName)      = 0;

 virtual CUserDialogHandle::UserDlgFeedbackType CreateMessageBox(
                                           const char *szText)   = 0;

 virtual ~IUserDialogFactory() {};
};
```

8.7. AbstractConfigurationStorage.h

This file describes an abstract interface to a settings management facility for external modules.

IUserDialogWatchdog

```
class IUserDialogWatchdog
{
public:
 virtual void ValueChanged(const char *szName) = 0;
};
```

IConfigurationStorage

```
class IConfigurationStorage
{
public:
 virtual void InstallWatchdog(IUserDialogWatchdog *pDialogWatchdog)   = 0;
 virtual void UninstallWatchdog(IUserDialogWatchdog *pDialogWatchdog) = 0;
 virtual void SendNotification(const char *szName) const              = 0;
 virtual const char *GetConfigurationValue(const char *szName,
                                           const char *szDefault)    = 0;
 virtual BOOL GetConfigurationValueBOOL(const char *szName,
                                           BOOL bDefault)            = 0;
 virtual int  GetConfigurationValueINT(const char *szName,
                                           int iDefault )            = 0;
 virtual void SetConfigurationValue(const char *szName,
                                    const char *szValue)             = 0;
 virtual void SetConfigurationValueBOOL(const char *szName,
                                        BOOL bValue)                 = 0;
 virtual void SetConfigurationValueINT(const char *szName,
                                        int iValue)                  = 0;
};
```