NEURO
CHECK
Industrial Vision Systems

Version 5.1
for Microsoft® Windows®
2000 / XP / Vista / 7

Programmer's Reference

Contents

Introduction

In addition to the extensive functionality contained as built-in, NeuroCheck provides the experienced user with three programming interfaces for extending the system. These interfaces allow you to integrate self-developed image processing or communication functions in NeuroCheck. In addition, you can run NeuroCheck as a software component in the background and provide the operator with an individually developed Windows application as user interface and control instance.

This *Programmer's Reference* describes the three programming interfaces which allow NeuroCheck to be enhanced by user-defined functionality. These interfaces are:

- The **plug-in interface** for user-defined check functions and menu items.
- The **custom communication interface** for user-defined (serial) communication protocols.
- The **OLE automation interface** for remote-control of NeuroCheck through other Win32 applications.

In addition, this document contains a description of the **XML format** of the NeuroCheck check routine and a **Quick Reference** for all interfaces presented.

Plug-In Interface

NeuroCheck's plug-in interface lets you integrate dynamic link libraries (DLLs) with user-defined functions for image processing and communication. Each DLL can contain an arbitrary number of self-developed check functions. These functions have full access to data objects created by NeuroCheck such as images or object features and are seamlessly integrated into the user interface. Furthermore, the DLL can extend the menu structure of NeuroCheck with customized menu commands.

NeuroCheck programmers typically use this interface for tasks like:

- control of special hardware (e.g. xy-table)
- database retrieval of external target values
- use of existing, specialized image processing know-how from self-developed libraries or third-party products
- starting external programs, e.g. for back-up purposes or other administrative functions

Plug-In-DLLs for NeuroCheck can be created using common software development tools. The typical programming language for this is "C" or "C++".

The documentation of the plug-in interface is structured as follows:

- The first chapter explains how to use a finished plug-in DLL in NeuroCheck.
- The second chapter describes the administration framework for plug-in functions provided by NeuroCheck and explains how the plug-in interface works and what operations are performed when a plug-in function is loaded and executed.

- The third chapter details the data types and declarations used for accessing internal NeuroCheck data structures and customized data structures in plug-in functions.
- The fourth chapter describes the use of the NeuroCheck API functionality in a plug-in DLL.
- The fifth chapter gives an overview of the sample plug-in DLLs included in the NeuroCheck installation.

Custom Communication Interface

This interface has been designed for the integration of communication drivers implemented as dynamic link libraries (DLL). Thus NeuroCheck can be remote-controlled by any type of communication hardware and can transmit result data using this hardware.

A typical example is the support of proprietary serial protocols for connecting to a PLC (programmable logic control) unit. Other examples might be the output of result data to a database or a quality management system, the implementation of a proprietary output file format or data transmission via TCP/IP. Also, basic control tasks like a conditional change of a check routine or a loop might be realized with this interface.

The usage and the implementation of a custom communication DLL is explained in the sixth chapter.

OLE Automation Interface

NeuroCheck supports OLE automation, the Windows standard for direct communication between programs. Thus NeuroCheck can be run as a server which is remote-controlled by a self-developed OLE controller program.

Such an OLE controller program can, for instance, load and run a complete NeuroCheck check routine. It also can access result values or modify parameter settings of the inspection task. The controller program thus can be used to provide an individual user interface for specific applications or to perform complex communication and control tasks.

Controller programs can be developed using common software development tools for Windows applications like, for example, Visual Basic®.

Chapter seven contains the complete description of the OLE automation interface of NeuroCheck. After an introduction to OLE in general it details and explains the available interface calls for NeuroCheck including source code extracts for both Visual Basic® and Visual C++®.

Check Routine in XML Format

NeuroCheck check routines can be exported in the human readable XML format. This facilitates a custom visualization of your check routine solutions for analysis and documentation purposes. Furthermore, it allows to monitor changes between check routines and the external manipulation or even automatic creation of check routines.

Chapter eight explains the XML format used for the check routine data and indicates possible applications. In addition, it gives hints and instructions for custom visualization using extended style sheets (XSL).

Quick Reference

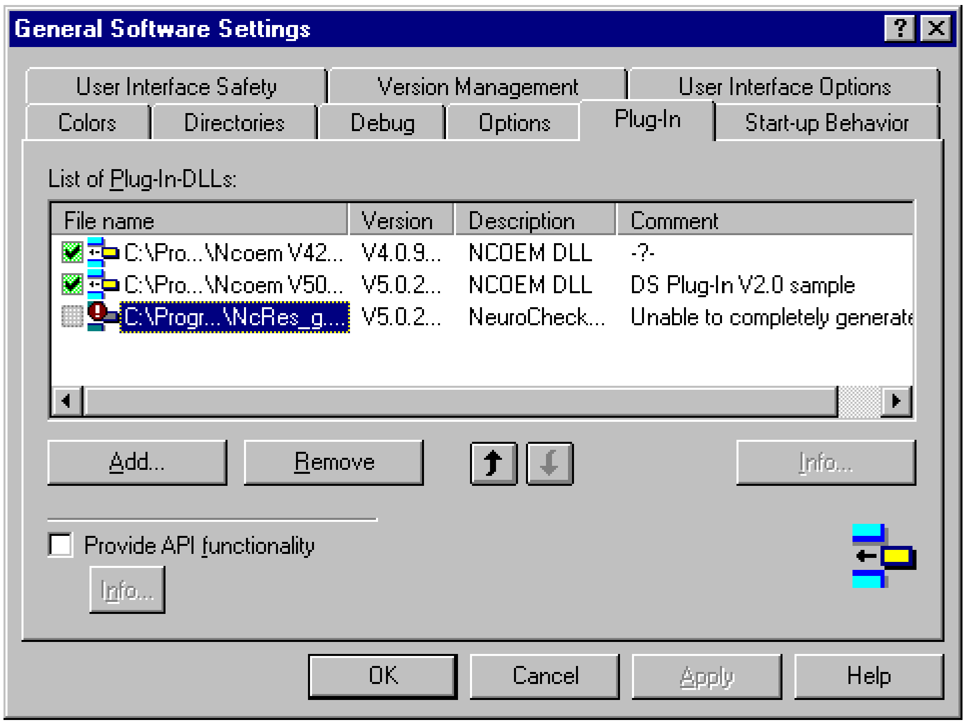
The ninth chapter finally represents a Quick Reference by summarizing syntax and parameters of all functions and data structures relevant for programming user extensions to NeuroCheck.

1 Using a Plug-In DLL in NeuroCheck

When realizing image processing applications it may happen that the main part of the inspection problem can be solved using the built-in standard functions of NeuroCheck, whereas a few requirements cannot be met. Often these requirements are so specific to the application that the cost and implementation overhead needed to integrate the solution into a standard software package are prohibitively large. The plug-in interface has been defined to allow such specific functionality to be added to NeuroCheck without changes to the main body of the program.
This chapter describes how to load and use a fully implemented plug-in DLL in NeuroCheck.

1.1 Loading a Plug-In DLL

Loading a plug-in DLL is initiated by choosing **Options** from the **System** menu. On page Plug-In of the General Software Settings dialog box clicking the Add button opens a file select dialog for selecting the DLL to be loaded. The plug-in is loaded dynamically.
A successfully loaded DLL will be indicated with the plug-in icon and a green check box.



File name	Version	Description	Comment
✓ C:\Pro...\Ncoem V42...	V4.0.9...	NCOEM DLL	-?-
✓ C:\Pro...\Ncoem V50...	V5.0.2...	NCOEM DLL	DS Plug-In V2.0 sample
C:\Progr...\NcRes_g....	V5.0.2...	NeuroCheck...	Unable to completely generate

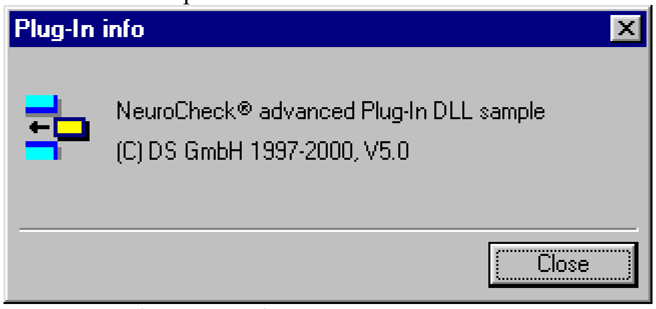
Figure 1: property page for loading a plug-in DLL into NeuroCheck

Error loading of the DLL will be indicated by a red icon. The comment column will contain Information about the error. If output of a global debug log file is activated on page Debug, details on the error will be written to the file `Nclog.txt`.
Each successfully loaded plug-in DLL can be activated or deactivated. A deactivated DLL will not be used, but remains loaded. Selecting a DLL and clicking the Remove button removes the DLL from the list.
The order of loading different DLLs depends on the order in the list of plug-in DLLs. This may be important if some plug-in DLLs depend on each other. The order of loading the DLLs can be changed with the arrow buttons.
Selecting a certain DLL in the list and clicking the Info button opens a message box with information about the DLL.

1.2 Information About a Loaded Plug-In DLL

After successfully loading a plug-in DLL the NeuroCheck Help menu will contain a new menu item **About Plug-In DLL**. Choosing this menu item displays a sub menu list with the names of all loaded plug-in DLLs. The names of not activated DLLs will be disabled.

Choosing the name of a activated DLL opens a message box with information about the DLL. The appearance of this message box is defined by the programmer of the DLL. Usually it will contain name, version and purpose of the DLL to distinguish between plug-in extensions specific to different visual inspection systems. The following figure shows the info dialog of the included sample DLL:


Plug-In info

NeuroCheck® advanced Plug-In DLL sample
(C) DS GmbH 1997-2000, V5.0

Figure 2: info dialog of included sample DLL

Successful loading of at least one plug-in DLL is also indicated in the status bar by the plug-in symbol, as shown in the following figure:

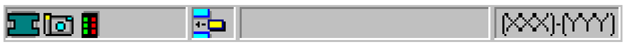
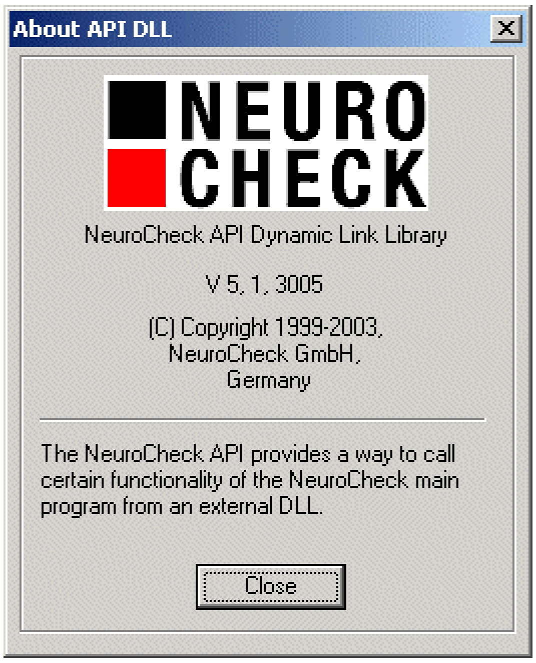


Figure 3: plug-in DLL indicator in status bar

Hint: Double-clicking the plug-in field in the status bar opens directly the page Plug-In of the General Software Settings dialog box.

1.3 Using the NeuroCheck API functionality

If at least one plug-in DLL has been loaded successfully, the NeuroCheck API functionality can be activated. Please note that this feature is only available, if NeuroCheck can find file `Ncapi.dll` in its installation path. Clicking the Info button opens a message box with information about the API functionality.



NeuroCheck API Dynamic Link Library

V 5, 1, 3005

(C) Copyright 1999-2003,
NeuroCheck GmbH,
Germany

The NeuroCheck API provides a way to call certain functionality of the NeuroCheck main program from an external DLL.

Figure 4: info dialog of NeuroCheck API DLL.

If the use of the NeuroCheck API functionality is activated, the plug-in symbol displayed in the status bar will change from yellow to green. Details on the NeuroCheck API functionality are listed in chapter „NeuroCheck API Functionality“.

1.4 Using Functions from a Plug-In DLL

1.4.1 Administration Frame for Plug-In Check Functions

Apart from a few restrictions NeuroCheck offers the same administration frame to functions from a plug-in DLL that is used for the built-in standard functions. The restrictions are:

- Plug-in functions cannot have a target value dialog. However, the parameter dialog can be used instead for target value input.
- It is not possible to duplicate the Apply-button functionality of updating the result display while the parameter dialog is being displayed, because the complicated interaction between main program and dialog required for this functionality would increase the administration overhead of the plug-in interface prohibitively.

The user will not notice any other differences between functions from a plug-in DLL and built-in functions. This means for example:

- Plug-in functions are added to a check routine like built-in functions, i.e. using the commands **New ► Append Check Function** resp. **New ► Insert Check Function** from the **Edit** menu. The functions are selected from their own page in the Select New Check Function dialog box. The dialog box displays a function description as well as hints for the usage of the function.
- Help topics describing the functions can be called by choosing **Info** from the **Check Function** menu or by clicking the Help button in the Select New Check Function dialog. NeuroCheck automatically redirects the help calls to a help file created by the programmer of the plug-in DLL.
- Plug-in functions can have a parameter dialog. The parameter copy and undo functionality is provided for these parameter dialogs in the same way as for the built-in functions.
- Plug-in functions access input data in the same way as built-in functions. The Input Data Configuration dialog operates in the usual manner.
- Plug-in functions are stored and loaded together with check routine like built-in functions. The programmer of the plug-in DLL does not need to provide his own file handling functionality.
- The names of the different result representations in the list boxes of the result display pane are customizable. If not specified, predefined names like Gray level image (In) are used.
- Plug-in functions can have their own graphical result display, allowing to visualize overlays (like drawing measuring lines into the source image) or custom data descriptions.
- Plug-in functions can participate on the standard NeuroCheck data output, i.e. output of bitmap files and data output to file or RS232 serial interface.

1.4.2 Using a Plug-In Check Function in a Check Routine

After a plug-in DLL has been successfully loaded, the Plug-In button in the Select New Check Function dialog box is enabled. Clicking the button, the functions of all activated plug-in DLLs are shown in a tree view. The functions are listed as sub nodes of the DLL they belong to. The function to be appended or inserted is selected as usual and integrated into the check routine.

For two plug in DLLs named `Ncoem V42.dll` and `Ncoem V50.dll` this will appear as in the following figure:

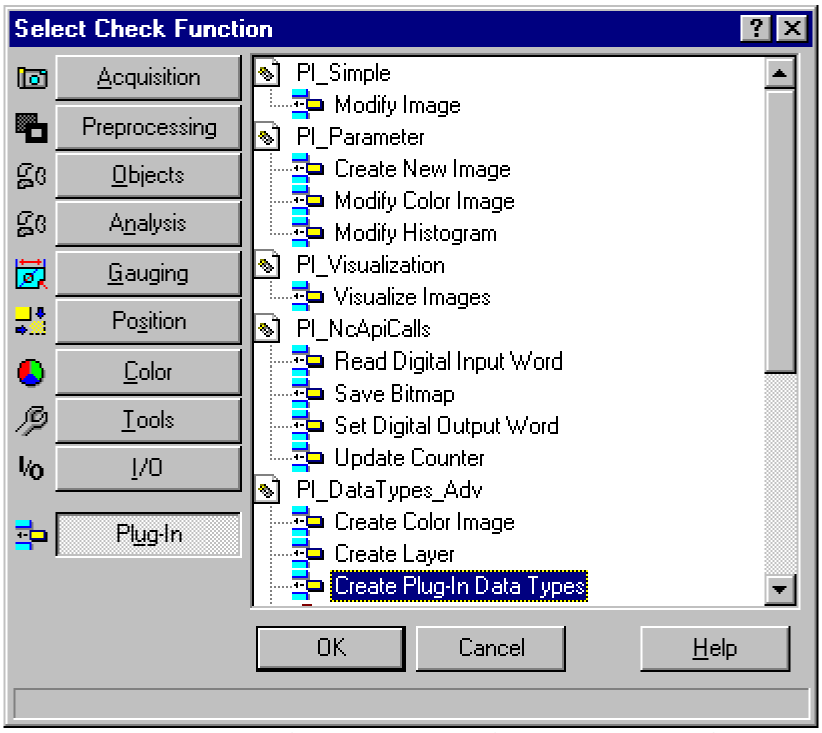


Figure 5: property page for plug-in check functions in NeuroCheck

1.5 The Plug-In Menu

In addition to user-defined check functions the plug-in DLL can define additional menu items, separately for manual and automatic mode. This allows the user to access special functionality, like initializing communication protocols, starting external programs etc.
The menu items are appended to the **Tools** menu as shown in the following figure for manual mode. The names of the individual menu items can of course be defined at will by the DLL programmer.

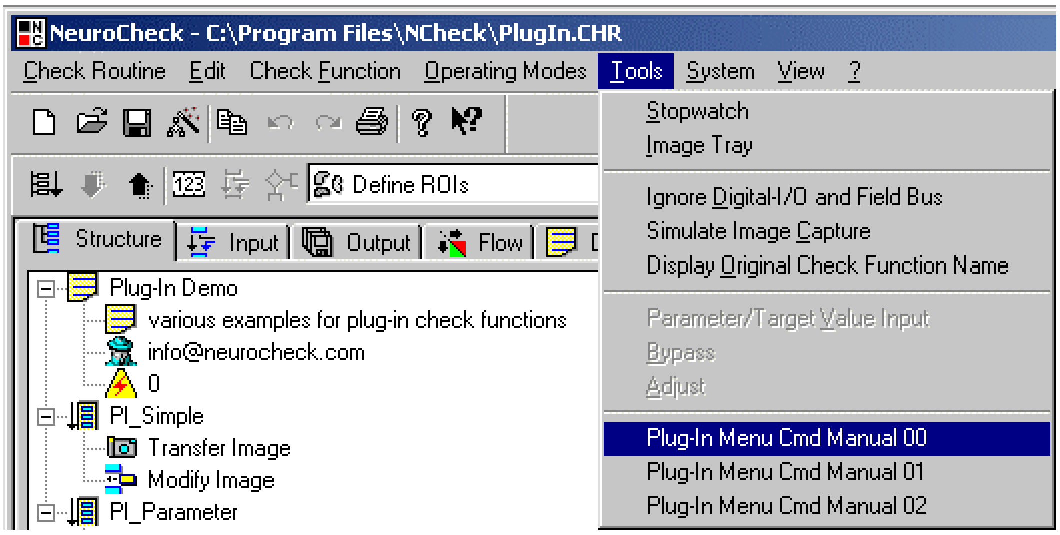

NeuroCheck - C:\Program Files\NCheck\PlugIn.CHR
Check Routine
Edit
Check Function
Operating Modes
Tools
System
View
?
Define ROIs
Structure
Input
Output
Flow
Plug-In Demo
various examples for plug-in check functions
info@neurocheck.com
0
PI_Simple
Transfer Image
Modify Image
PI_Parameter
Stopwatch
Image Tray
Ignore Digital-I/O and Field Bus
Simulate Image Capture
Display Original Check Function Name
Parameter/Target Value Input
Bypass
Adjust
Plug-In Menu Cmd Manual 00
Plug-In Menu Cmd Manual 01
Plug-In Menu Cmd Manual 02

Figure 6: plug-in menu for manual mode in NeuroCheck

2 Interface Description

This chapter describes the individual elements of the plug-in interface and the operations taking place during initialization, use and deinitialization of a plug-in DLL and its functions.

2.1 Interface Structure

To enable NeuroCheck to use the functions contained in a plug-in DLL certain conventions have to be adhered to. Every plug-in DLL has to contain certain administration functions accessible from within NeuroCheck with a specific name.
This section will first give an overview of the administration frame for the DLL and its functions. Subsequent sections will cover the individual administration functions in detail.

2.1.1 Managing a Plug-In DLL

Administration of a plug-in DLL in NeuroCheck consists of the following operations:

1. Upon program startup or upon loading the DLL on page Plug-In of the General Software Settings dialog box the DLL is loaded into the memory area of the NeuroCheck process and is initialized.
2. NeuroCheck calls routine `PI_GetNumberOfFcts()`. This routine is declared within NeuroCheck and has to be provided by every plug-in DLL. NeuroCheck allocates a control data structure for each of the plug-in check functions.
3. For each plug-in check function NeuroCheck calls routine `PI_GetFctCaps()` which returns a function info structure containing information required by NeuroCheck to handle the plug-in check function.
4. If exported from the plug-in DLL, NeuroCheck then calls routine `PI_GetNumberOfDataTypes()`. NeuroCheck allocates a control data structure for each of the plug-in data types.
5. For each data type NeuroCheck calls routine `PI_GetDataTypeDesc()` which returns a type description structure containing information required by NeuroCheck to handle the plug-in data type.
6. The plug-in check functions and data types can then be used like built-in functions and data types with the exception, that plug-in data types can only be used by plug-in check functions. Every call to a plug-in function causes data to be exchanged between NeuroCheck and the DLL.
7. Before exiting NeuroCheck removes the plug-in DLL from memory.

The following figure illustrates this administration framework.

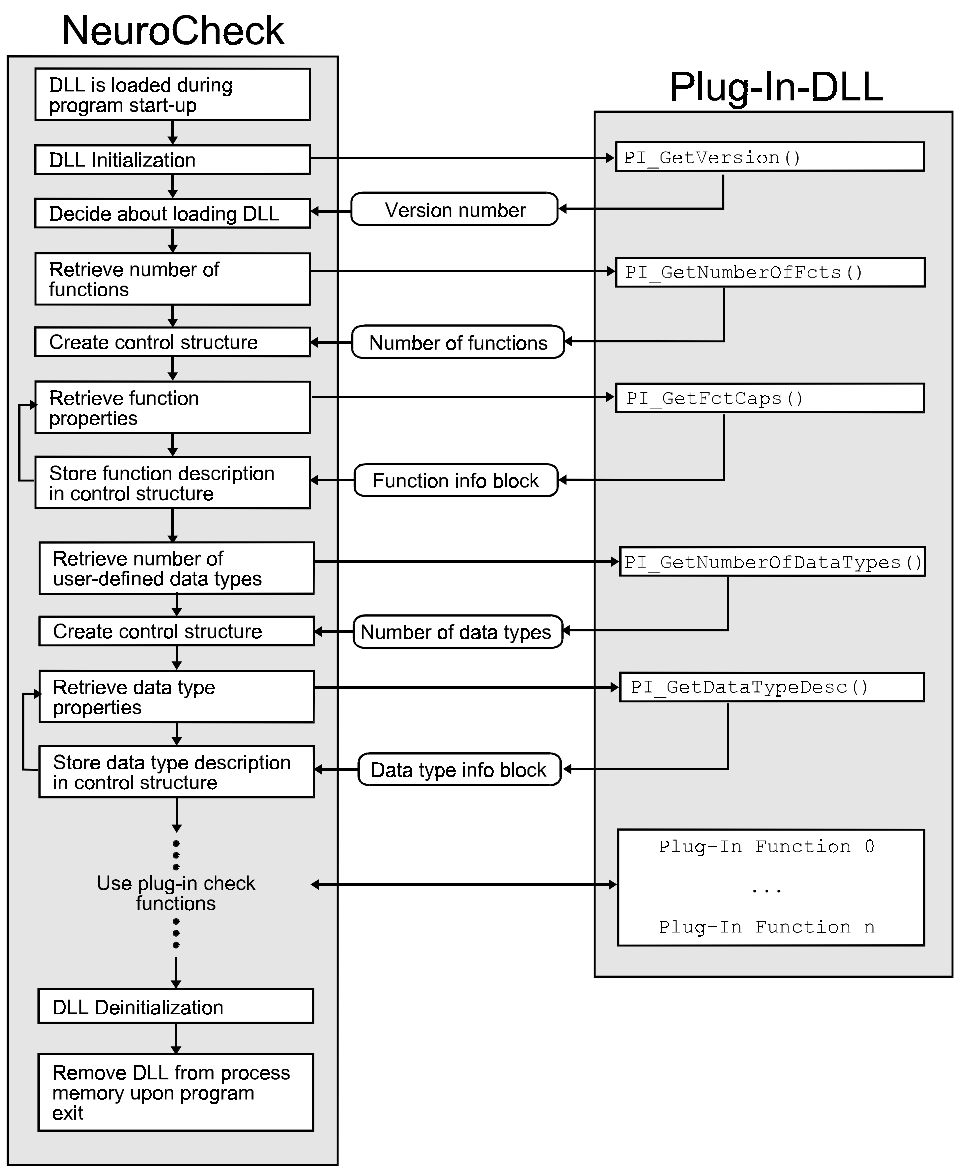

NeuroCheck
DLL is loaded during program start-up
DLL Initialization
Decide about loading DLL
Retrieve number of functions
Create control structure
Retrieve function properties
Store function description in control structure
Retrieve number of user-defined data types
Create control structure
Retrieve data type properties
Store data type description in control structure
Use plug-in check functions
DLL Deinitialization
Remove DLL from process memory upon program exit
Version number
Number of functions
Function info block
Number of data types
Data type info block
Plug-In-DLL
PI_GetVersion()
PI_GetNumberOfFcts()
PI_GetFctCaps()
PI_GetNumberOfDataTypes()
PI_GetDataTypeDesc()
Plug-In Function 0
...
Plug-In Function n

Figure 7: administration framework for plug-in DLLs in NeuroCheck

2.1.2 Managing Plug-In Check Functions

Administration of a plug-in check function consists of the following operations:

1. Inserting a plug-in check function.
2. Setting parameters of the check function.

3. Executing the check function.
4. Using the output possibilities of the check function.
5. Removing the check function from the check routine.

For every check function the plug-in DLL has to provide a dedicated routine for each of these operations. These functions are explained in detail in section „Structure of Plug-In Functions“. This section will explain only the services provided by the main program for the plug-in check functions.

Inserting a Plug-In Check Function

When a user-defined check function is inserted into (or appended to) a check routine, NeuroCheck retrieves information about the function from its info structure. This information comprises e.g.:

- Function name.
- Size of parameter area.
- Function addresses.

NeuroCheck allocates memory for the parameter area and executes the initialization routine of the plug-in check function. This initialization routine can provide services like setting default parameters, opening communication channels, initialize hardware etc. The following figure illustrates the process. The solid arrow going out from the initialization routine indicates that the routine is actually executed at this point, for example to set parameter values.

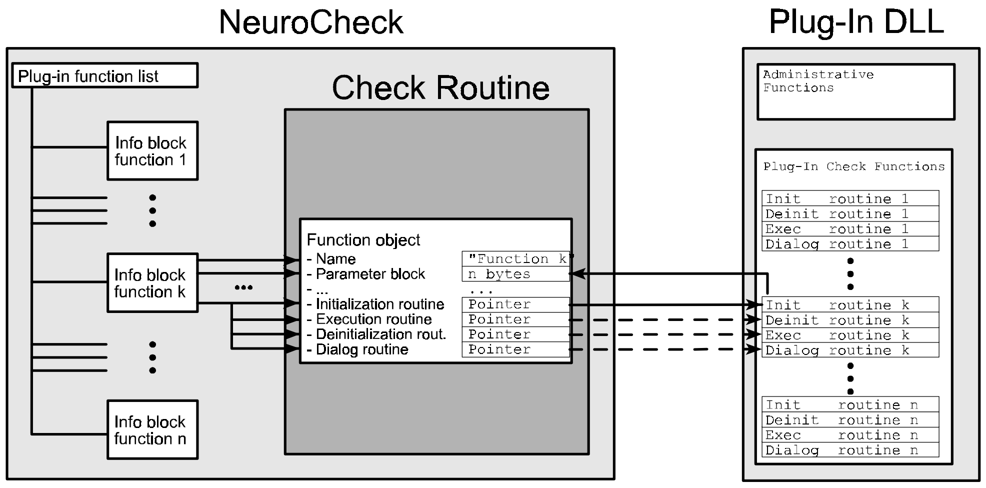


Figure 8: initialization of plug-in check function

Setting Plug-In Check Function Parameters

If the plug-in check function owns a parameter area and a dialog routine NeuroCheck will call this routine under the same conditions as for a built-in function, e.g. upon choosing **Parameters** from the **Check Function** menu. Dialog display, parameter value update and management of the internal structure of the parameter area are the business of the DLL alone. Using an unstructured buffer area NeuroCheck provides the one-step undo function existing for built-in functions. The button will revert the last changes made to the function parameters. In the same way NeuroCheck implements the parameter copy functionality. Validation of the copied parameter block is again the business of the plug-in DLL.

Executing a Plug-In Check Function

Whenever NeuroCheck encounters a plug-in check function in the course of a check routine, it will call the execution routine of the plug-in check function. The current context information (e.g. about the current operating mode), the parameter block and pointers to all required input data objects are passed to the function, according to the specifications in the function info structure.
The execution routine of a plug-in check function can create data objects of the types image, layer of ROIs, histogram and measurement list, if the function info structure registered these with NeuroCheck. In addition, it can create output data objects of same plug-in data type registered by its plug-in DLL. Objects of plug-in data types can only be passed to other plug-in functions of the same plug-in DLL whereas NeuroCheck data objects can be used by any check function. The execution routine will receive pointers to adresses of such data objects, can allocate memory for the data objects and return the adresses to NeuroCheck. Conventions for memory allocation are described below.

Using the Output Possibilities of a Plug-In Check Function

If registered in the function info structure, NeuroCheck will update the plug-in check function's graphical output. Each function can have several user-defined „Views". Plug-in check function also can take part in the standard data output of NeuroCheck, i.e. output of result values to file or serial interface.

Removing a Plug-In Check Function

Before removing a plug-in check function NeuroCheck calls its deinitialization routine for clean-up of any initialized structures, e.g. communication channels.

2.2 Administrative Functions

To organize communication between NeuroCheck and the plug-in DLL certain administrative functions are required. Their names, declarations and calling conventions are defined within the main program and must not be changed!

2.2.1 Version Information

Every plug-in DLL has to inform NeuroCheck about the interface version for which it has been compiled. During DLL loading, NeuroCheck calls this function and compares the value returned with the internal interface version information. If the value returned by the DLL function does not match any of the plug-in interface versions provided by NeuroCheck, it will refuse to load the DLL and you will have to recompile it after implementing the necessary changes.
The function declaration is:

```
extern "C" unsigned int WINAPI PI_GetVersion(void)
```

Please note that NeuroCheck will recognize three plug-in interface versions

- Interface version 400 for compatibility with DLLs developed for NeuroCheck 4.x
- Interface version 500 for compatibility with DLLs developed for NeuroCheck 5.0 and 5.1 up to SP4.
- Interface version 510 for DLLs developed for NeuroCheck 5.1 SP5 or later according to the interface description in this reference.

Please note that the differences between 500 and 510 are neglectible. Therefore, upgrading a DLL from interface version 500 to 510 simply means to re-compile with the header file `PI_Types.h`. Check routines containing plug-in functions created for interface version 500 still can be loaded with the new DLL of interface version 510.
The plug-in samples included with the NeuroCheck setup will always contain a function returning the correct number for the respective NeuroCheck version.

2.2.2 Info-Dialog

Every plug-in DLL has to display an information dialog. This function is called from within NeuroCheck by clicking the Info button on page Plug-In of the General Software Settings dialog box, or by choosing the DLL name from the sub menu of **About Plug-In DLL** from the **Help** menu. Its declaration is:

```
extern "C" void WINAPI DllInfo(HWND hwndMain)
```

Its single function parameter is the handle for the NeuroCheck main window.

2.2.3 Help File

A plug-in DLL programmer can provide the user of his DLL with a dedicated help file in WinHelp or HtmlHelp format. After loading the DLL NeuroCheck calls function

```
extern "C" BOOL WINAPI PI_GetHelpFilePath(LPSTR lpszPath)
```

to retrieve filename and path of the help file to be used for all help calls concerning plug-in functions. In `lpszPath` the address of a buffer with size `_MAX_PATH` is passed to the function so that the function can copy any legal path name into this buffer. NeuroCheck will decide upon the file extension whether to call WinHelp (*.hlp) or HtmlHelp (*.chm). If no help file has been provided for the plug-in DLL the function should return `FALSE`. NeuroCheck will then ignore all help inquiries concerning plug-in functions.

2.2.4 Menu Command Handler

The plug-in DLL can implement additional menu commands for the **Tools** menu in manual and automatic mode. This is explained in detail in section „The Plug-In Menu“. NeuroCheck reads the menus from the DLL resources. When one of the custom menu commands is selected during runtime, NeuroCheck calls function

```
extern "C" void WINAPI PI_MenuCommand(
                          HWND hwndMain,
                          unsigned int uiCmdId)
```

with the handle of the main application window and the ID of the selected menu item as parameters.

2.2.5 Information about Plug-In Check Functions

NeuroCheck requires information about the properties of the check functions defined in the DLL to be able to utilize them. Two functions serve this purpose:

```
extern "C" unsigned int WINAPI PI_GetNumberOfFcts(void)
```

is called immediately after loading the DLL and must return the number of check functions contained in the DLL. NeuroCheck uses this number to allocate the required number of function info blocks.
The following function returns information about the capabilities of individual plug-in check functions.

```
extern "C" BOOL WINAPI PI_GetFctCaps(
                       unsigned short int uiIndex,
                       sPI_FCT_DESC* const psFctDesc)
```

It is called in a loop after `PI_GetNumberOfFcts()` once for each function. The loop counter is passed in `uiIndex` (counted from 0 up to the return value of `PI_GetNumberOfFcts()` minus 1). In `psFctDesc` NeuroCheck passes the address of the allocated function info block for the current function. Function `PI_GetFctCaps()` has to fill in the information about the check function indicated by the index into the members of this info structure.

2.2.6 Information about Plug-In Data Types

Plug-in check functions may create and use user-defined data objects. The information about the data types of these objects is retrieved from the DLL in a similar way as the check function capabilities. The two routines serving this purpose are:

```
extern "C" unsigned int WINAPI PI_GetNumberOfDataTypes(void)
```

is called after the iteration over the plug-in check function capabilites. It must return the number of user-defined data types used by the plug-in check functions. NeuroCheck uses this number to allocate the required number of data type info blocks.
The following function returns information about the properties of individual plug-in data types.

```
extern "C" BOOL WINAPI PI_GetDataTypeDesc(
                       unsigned short int uiIndex,
                       sPI_TYPE_DESC* const psTypeDesc)
```

It is called in a loop after `PI_GetNumberOfDataTypes()` once for each type. The loop counter is passed in `uiIndex` (counted from 0 up to the return value of `PI_ GetNumberOfDataTypes()` minus 1). In `psTypeDesc` NeuroCheck passes the address of the allocated info block for the current type. Function `PI_GetDataTypeDesc()` has to fill in the information about the data type indicated by the index into the members of this info structure.

2.3 Memory Management

2.3.1 Output Data Objects of NeuroCheck Data Types

Plug-in check functions can create output data objects of the NeuroCheck data types (image, layer of regions, histogram, measurement list) just like built-in functions. Data objects created by build-in functions are deleted after the check routine has been completely executed. To enable NeuroCheck to delete objects dynamically allocated by plug-in check functions in the same way (and to free the plug-in DLL programmer from managing these objects himself) all objects of a NeuroCheck data type must be allocated using the virtual heap API function:

```
extern "C" VOID * WINAPI VirtualAlloc(
                          LPVOID  lpAddress,
                          DWORD   dwSize,
                          DWORD   flAllocationType,
                          DWORD   flProtect)
```

`VirtualAlloc` initializes all elements of the allocated memory block to 0.

Important: NeuroCheck de-allocates all dynamical output data returned by the execution routine, independently of the result of the routine. If you de-allocate data objects (with `VirutalFree`) already inserted in the NeuroCheck container structure, always assign a `NULL` value to the freed pointer.

2.3.2 Output Data Objects of Plug-In Data Types

In addition to the NeuroCheck data types, each plug-in DLL can register its own data types. Only plug-in check functions registered by the same DLL may create output data objects of the plug-in data types. These output data objects are allocated by the execution routine of the plug-in check function. The plug-in DLL programmer must manage these objects himself which implies that he can choose the way of memory allocation for the data objects.
The objects can be passed on to other plug-in functions which belong to the same DLL. For efficient use, the user-defined data objects are passed as pointers and will only be interpreted inside the plug-in functions using them. Since NeuroCheck cannot interpret these objects, it is not able to delete their dynamically allocated data structure. For freeing the allocated memory, NeuroCheck calls the destroy routine of the plug-in data type passing the pointer to the data structure to be de-allocated. For de-allocation, the plug-in DLL must use the operator matching the allocation operator previousely used to create the object.

2.3.3 Custom Result Views

Each plug-in check function may create its own result view displayed by NeuroCheck in the Result View pane of the function in manual or test mode or in its visualization window in automatic mode. For retrieving the current output of a custom view, NeuroCheck calls the plug-in check function's view create routine. This routine returns a handle to a bitmap to be displayed. The plug-in programmer must manage the bitmap objects himself. For convenience, he may use the NeuroCheck API functionality provided for bitmap management. When the bitmap handle can be released, NeuroCheck calls the plug-in check function's view destroy routine. For de-allocation, the plug-in DLL must use the operator matching the allocation operator previousely used to create the object.

2.3.4 Data Output to File or RS232 Serial Interface

For output to file or RS232 serial interface, a data output routine can be registered for each plug-in data type. Calling this routine, a container is passed for returning a data sequence which NeuroCheck will write to file or serial interface. This data sequence must be allocated by the plug-in DLL using `VirtualAlloc` and will be de-allocated by NeuroCheck.

2.3.5 Note

The above applies to output data objects collected in the NeuroCheck data pool only. Functions inside the plug-in DLL may of course allocate dynamical data for internal use using the `new` operator and free it using `delete`. Just make sure that all memory allocated inside the DLL is released again. Memory allocated inside the initialization routine may be freed in the deinitialization routine, because both routines will be executed only once during the lifetime of the plug-in check function, but memory allocated inside the execution or parameter dialog routines must be freed inside the same routine, because these may be executed arbitrarily often.

2.4 The Plug-In Menu

After loading a plug-in DLL NeuroCheck reads the definition of a plug-in menu from the resource portion of the DLL. Menu items of such a menu are appended to the **Tools** menu. NeuroCheck distinguishes between a menu for manual mode and a menu for automatic mode. Refer to the description of the included sample DLL for implementation details.

2.4.1 Manual Mode Menu

The manual mode menu has a predefined resource ID of `0x6000`. Other IDs may interfere with internal functions of NeuroCheck.

2.4.2 Automatic Mode Menu

The automatic mode menu has a predefined resource ID of `0x6001`. Other IDs may interfere with internal functions of NeuroCheck.

2.4.3 Menu Items

The number of user-defined menu items is restricted to 256. The resource IDs of the menu items have to be within a range from `0xD000` to `0xD100`. Other IDs may interfere with internal functions of NeuroCheck.

2.4.4 Menu Handler

When one of the custom menu commands is selected during runtime, NeuroCheck calls function

```
extern "C" void WINAPI PI_MenuCommand(HWND hwndMain,
                                      unsigned int uiCmdId)
```

In `hwndMain` the handle of the main application window is passed to the function. `uiCmdId` receives the resource ID of the selected menu item.

2.5 Structure of a Plug-In Check Function

This section describes the possible capabilities of plug-in check functions and the declarations of the routines making up a plug-in check function. A plug-in check function is implemented through six distinct routines:

1. Initialization routine.
2. Execution routine.
3. Deinitialization routine.
4. Parameter dialog routine.
5. Custom view create routine.
6. Custom view destroy routine.

The names of these functions are unimportant, because NeuroCheck calls them through function pointers in the function info structure. The declarations though are fixed.
Apart from the parameter dialog routine and the two visualization routines, all routines have to exist for every plug-in check function.

2.5.1 Capabilities of Plug-In Check Functions

Plug-in check functions can have a parameter area of (in principle) unlimited size. NeuroCheck manages this as an unstructured memory block, i.e. as a byte array. Only the DLL routines themselves can change the values inside this block so that the programmer of a plug-in DLL can rely on NeuroCheck never to interfere with the inner workings of the parameter area. There are only two ways for NeuroCheck to change the parameter area and both of them make use of values set within the DLL routines:

1. The undo function copies the most recent version of the parameter area into the current parameter block.
2. The **Copy / Paste Parameters** functions copy a parameter block from one function into the block of another function. The unique identification numbers required for plug-in check functions ensure that only compatible blocks are copied.

Plug-in check functions can use up to five input data objects of the various NeuroCheck data types (image, layer of regions, histogram, measurement list) or some plug-in data type registered by the function's plug-in DLL. It can create up to five dynamical output data objects of these types (see section „Memory Management“ about conventions for the administration of these data objects). Some restrictions apply to the operations on the dynamical input data objects allowed within a plug-in check function. See chapter „Data Types and Objects“ for details.

2.5.2 Initialization Routine

The initialization routine of a plug-in check function has to be declared as follows (excepting the actual name of course):

```
BOOL WINAPI InitFunction(
                              sPI_CONTEXT_INFO* const psContext,
                              void* const pParameter);
```

In `psContext` NeuroCheck passes information about the current context of the plug-in check function. This might be useful because plug-in check functions may have to behave differently for an other context. For instance, a function may have different behaviour in automatic and in manual mode. The following structure is used by NeuroCheck for context information:

```
typedef struct
{
    int      iSingleCheckIndex;
    int      iCheckFunctionIndex;
    int      iOID;
    int      iMode;
}   sPI_CONTEXT_INFO;
```

The structure elements have the following meaning:

Element	Description
`iSingleCheckIndex`	Current index of single check the plug-in check-function belongs to. For start and end actions, a special negative index is given.
`iCheckFunctionIndex`	Current index of plug-in check function within single check.
`iOID`	Object identification number (OID) of plug-in check function. This number is unique within the check routine.
`iMode`	Current operating mode. Can have one of the following values, declared as symbolic constants in `pi_types.h`: `NC_MODE_MANUAL` manual mode. `NC_MODE_TEST` test mode. `NC_MODE_LIVE` live mode. `NC_MODE_AUTOMATIC` automatic mode. `NC_MODE_AUTOCONFIG` configure automatic screen mode. `NC_MODE_INTRO` HTML introduction mode.

All elements of the structure have to be considered „read-only". Please note that this also applies to all other functions which are passed context information.
In `pParameter` NeuroCheck passes the address of a memory block reserved for the function parameters according to the size given in the function info block. `pParameter` is passed as *constant pointer* which means that the pointer itself must not be changed. Nevertheless, it is well possible to change the contents of the memory block the pointer is pointing to after casting it to the appropriate type. If no memory block has been reserved due to size 0 given in the function info block, the pointer will be `NULL`. Please note that this also applies to all other functions which are passed the parameter memory block.
The initialization routine is called whenever NeuroCheck creates an instance of the respective plug-in check function, i.e. when this plug-in check function is appended to or inserted into a check routine, or when a check routine is loaded which contains this plug-in check function. It will typically be used for setting default parameters.

NeuroCheck evaluates the function return value. If the function returns `FALSE` NeuroCheck assumes that the initialization failed. The check function will not be added to the check routine (or the check routine will not be loaded).

Hint: Since the initialization routine is executed when the check function is first added to a check and also when the check routine is loaded from file, it is useful, to indicate in the parameter area, whether parameters have already been edited for this function.
A simple method would be to use the first byte of the parameter area as an initialization flag. Before the check function is actually added to the check routine the whole parameter block will be filled with zeroes. The initialization should set default parameters values only if the first byte is indeed zero and then set the byte to a different value. This ensures that altered parameters will not be inadvertently reset to default values. It might also be useful to encode a version information, e.g. in the second byte, in case of changes to the structure of the parameter area between two versions of a plug-in DLL. Please note, that this only works, if the size of the memory block reserved for the parameter set remains identical between two versions of a plug-in check function because NeuroCheck will fail to load a plug-in check function from a check routine if it had been saved with a differently sized parameter areas.

2.5.3 Execution Routine

The execution routine of a plug-in check function has to be declared as follows:

```
BOOL WINAPI ExecuteFunction(
              sPI_CONTEXT_INFO* const psContext,
              void* const pParameter,
              void* const * const ppvDynInput,
              void*       * const ppvDynOutput);
```

In `psContext` NeuroCheck passes the function's current context information, in `pParameter` the address of its parameter block. `ppvDynInput` is a container of the input data objects required in the function info block, i.e. it is an array of the addresses of the objects. It has at most five valid members. For a function without dynamical input data objects no object will be valid. For a function requiring one input data object, `ppvDynInput [0]` will contain the address of this object, the other members will not be valid, etc. The addresses have to be casted inside the function to the respective input data type: either a NeuroCheck data type (image, histogram, region layer, measurement list) or a user-defined data type registered in the plug-in DLL containing the plug-in check function.
Both the address of the container and the addresses of the input objects are passed as *constant* pointers which means that they must not be altered or de-allocated. However, it is possible to alter the value of a member of an input data object inside the execution routine unless this member is read-only.
Similarly, `ppvDynOutput` is a container for the possible output data objects. Please note, that the addresses of the output object pointers are not constant, i.e. the execution routine creates the required output objects and assignes their addresses to the elements of `ppvDynOutput`. See the example DLL for more information.
This routine is executed whenever NeuroCheck encounters the respective plug-in check function during a check routine run. The return value of the function determines the further behavior of NeuroCheck. If the function returns `FALSE`, the current individual check will be

terminated with status `not O.K.` Every plug-in check function can be used as a decision function in this way.

2.5.4 Deinitialization Routine

The deinitialization routine of a plug-in check function has to be declared as follows:

```
void WINAPI DeInitFunction(
             sPI_CONTEXT_INFO* const psContext,
             void* const pParameter);
```

In `psContext` NeuroCheck passes the function's current context information, in `pParameter` the address of its parameter block.
This routine is executed whenever NeuroCheck removes an instance of the pertaining plug-in check function, i.e. when deleting the function from a check routine, deleting the whole check or closing the check routine. If the initialization routine performed any activities affecting the whole DLL, program or system (like allocating dynamic memory or opening communication channels) the effects can be reverted here. Therefore the function is allowed access to the parameter area in order to be able to react to the current parameter set. Changes to the parameter set will have no effect for this function.

2.5.5 Parameter Dialog Routine

The parameter dialog routine of a plug-in check function has to be declared as follows:

```
BOOL WINAPI ParameterDialog(
             HWND hwndApp,
             sPI_CONTEXT_INFO* const psContext,
             void* const pParameter,
             void* const * const ppvInputData);
```

In `psContext` NeuroCheck passes the function's current context information, in `pParameter` the address of its parameter block. In `hWndApp` NeuroCheck passes a handle for the parent window. `ppvInputData` is an array of the addresses of the required input data objects equivalent to the argument `ppvDynInput` of the execution routine. The function handling the dialog has to cast these addresses to pointers of the correct type for the respective input data object. The input objects should be considered read-only, i.e. any changes will have no effect. See the example DLL for more information.
This routine is executed whenever one of the events occurs which would open the parameter dialog of a build-in check function in NeuroCheck , e.g. choosing **Parameters** from the **Check Function** menu.

2.5.6 Custom Visualization Routines

For custom visualization, a plug-in check function must register two routines, the view create routine and the view destroy routine. They have to be declared as follows:

```
BOOL WINAPI ViewCreate(
             sPI_CONTEXT_INFO* const psContext,
             sPI_VIEW_INFO* const psViewInfo,
             HANDLE* hView,
             void* const pParameter,
                void* const * const ppvInputData,
                void* const * const ppvOutputData);

BOOL WINAPI ViewDestroy(
                sPI_VIEW_INFO* const psViewInfo,
                HANDLE* hView);
```

For the view create routine, NeuroCheck again passes the function's current context information in `psContext`, and the address of its parameter block in `pParameter`. In addition, both visualization routines are passed specific information for visualization in `psViewInfo`. The following structure is used by NeuroCheck for visualization information:

```
typedef struct
{
    unsigned int      uiViewIndex;
    unsigned int      uiViewType;
    unsigned int      uiAddInfo;
    int               iExecuteErrorCode;
}   sPI_VIEW_INFO;
```

The structure elements have the following meaning:

Element	**Description**	
`uiViewIndex`	Index of custom view registered for plug-in check function.	
`uiViewType`	Type of view. Can have only one possible value, declared as symbolic constant in `pi_types.h`:	
	`PI_VIEW_BITMAP`	handle to bitmap is returned.
`uiAddInfo`	Additional informaion. Value indicates current zoom factor of result view. Can have one of the following values, declared as symbolic constants in `pi_types.h`:	
	`NC_ZOOM_10`	10 % of original image size.
	`NC_ZOOM_25`	25 % of original image size.
	`NC_ZOOM_50`	50 % of original image size.
	`NC_ZOOM_100`	100 % of original image size.
	`NC_ZOOM_200`	200 % of original image size.
`iExecuteErrorCode`	Error code for most recent call to execution routine of the plug-in check function. Can have one of two values, declared as symbolic constants in `pi_types.h`:	
	`NC_EXE_OK`	execution result was "O.K.".
	`NC_EXE_NOK`	execution result was "not O.K.".

All elements of the structure have to be considered „read-only“.

In `hView` the view create routine returns a handle to a bitmap which will be displayed by NeuroCheck in the Result View pane of the plug-in check function in manual mode or in its visualization window in automatic mode. This handle will be passed at the succeeding call to the view destroy routine when the handle can be released.
In addition, in `ppvInputData` and `ppvOutputData` the current input and output data objects of the most recent execution of the plug-in check function are passed to the view create routine. Both input and output objects should be considered „read-only“. Please note

that if the execution result had been "not O.K.", the output data objects may not be valid and thus be assigned `NULL` pointers.
In manual mode the view routine is called directly after a call to the execution routine, or upon changing settings of the currently displayed view in the Result View pane, e.g. view index or zoom factor. Please note that in manual mode NeuroCheck allows subdivision of the Result View pane into two horizontal panes and thus actually provides two views. So directly after a call to the execution routine, the view routine is always executed twice.
In automatic mode, the view routine is called immediately after execution of the plug-in check function once for each of the function's visualization windows.
The view create routine is allowed access to the plug-in check function's context, parameter set and input and output data. Thus, the function can react to the current settings and perform a custom visualization. This may include visualization of input and output data, of final or intermediate results, or even of specific messages.

2.6 The Function Info Block

Function `PI_GetFctCaps(uiIndex, psFctDesc)` has to fill the function info block whose address is passed in `psFctDesc` for every possible index `uiIndex` with the characteristics of the respective plug-in check function. This section describes the structure of the function info block and the meaning of its members.

2.6.1 Function Info Block Declaration

The function info block is declared as follows (detailed explanations of the individual sections follow below):

```
typedef struct
{
    // version control
    unsigned int                uiStructSize;

    // data
    unsigned int                uiFunctionId;
    unsigned int                uiHelpContext;
    unsigned int                uiNumOfInputData;
    unsigned int                uiNumOfOutputData;

    unsigned int                uiSizeOfParameter;
    unsigned int                uiNumOfViews;
    unsigned int                uiDataOutput;

    // pointer
    // - data
    int*                        piTypeOfInputData;
    int*                        piTypeOfOutputData;
    int*                        piMaskFileOutput;
    int*                        piMaskRs232Output;

    // - strings
    LPSTR                       lpszFunctionName;
    LPSTR                       lpszDescription;
    LPSTR                       lpszHintPos;
    LPSTR                       lpszHintNeg;
    LPSTR*                      alpszCustomViewNames;
```

```
        LPSTR*                         alpszInputViewNames;
        LPSTR*                         alpszOutputViewNames;

        // - functions
        PFNPlugInFctInitialize         pfnInitialize;
        PFNPlugInFctExecute            pfnExecute;
        PFNPlugInFctUnInitialize       pfnUnInitialize;
        PFNPlugInFctParameterDlg       pfnParameterDlg;
        PFNPlugInFctViewCreate         pfnViewCreate;
        PFNPlugInFctViewDestroy        pfnViewDestroy;
    }
        sPI_FCT_DESC;
```

2.6.2 General Information

The data area of the function info block contains the following elements:

Element	Description
`uiStructSize`	NeuroCheck allocates the info block and initializes each element to 0 with the exception of its first element `uiStructSize`. This element contains the size of the info block as returned by the `sizeof()` function. The value of `uiStructSize` should be considered read-only and can be used inside the plug-in DLL to verify that NeuroCheck and the plug-in DLL use the same declaration of the `sPI_FCT_DESC` structure. If the value of `uiStructSize` and the return value of `sizeof(sPI_FCT_DESC)` are not equal, there is a version conflict between the plug-in interface provided by NeuroCheck and the one expected by the DLL.
`uiFunctionId`	Unique identification number for the plug-in function. NeuroCheck uses this number for various purposes, among them identifying functions when loading a check routine. For this reason plug-in check functions inside a single plug-in DLLs must not use identical function IDs. **NOTE:** function IDs from `0xE0000000` upwards are reserved for use by NeuroCheck GmbH and partners. This leaves more than 3.75 * 10^9 entries free for customer use, ample space even for organizing the most extensive projects.
`uiHelpContext`	The context number NeuroCheck will pass to Winhelp when you choose **Info** from the **Check Function** menu or the Help button in the Select New Check Function dialog for a plug-in check function.
`uiNumOfInputData`	Number of expected input data objects. A plug-in check function can use up to `PI_MAXDYNDATA` input data objects. This symbolic constant is defined in `pi_types.h` with a value of `5`.

`uiNumOfOutputData`	Number of created output data objects. A plug-in check function can create up to `PI_MAXDYNDATA` output data objects. This symbolic constant is defined in `pi_types.h` with a value of `5`.
`uiSizeOfParameter`	Size of function parameter area. NeuroCheck treats the parameter array as an unstructured byte array of this size. Please note that changes to the size of the parameter area are crucial. NeuroCheck will not be able to load a plug-in check function from a check routine saved with a different size of the parameter area than expected while loading. If further parameters can be expected in future versions of a plug-in check function, it is advisable to reserve more than the currently needed memory space, giving the chance of maintaining downward compatibility. If the size of the parameter area indeed needs to be changed, it is recommended also to alter the function identification number.
`uiNumOfViews`	Number of user-defined views of the plug-in check function. If the element is larger than 0, the plug-in check function must provide a view routine and a name for each of the views.
`uiDataOutput`	Switch indicating whether the plug-in intends to take part in the standard NeuroCheck data output processes. The element should be 1 when the plug-in function provides standard output capabilities, it should be 0 if it does not provide any output capability.

2.6.3 Input / Output Data Object Description

The pointer area of the function info block contains two pointers which must be assigned to static arrays with at least `uiNumOfInputData` resp. `uiNumOfOutputData` elements each stating the types of the expected input and created output data objects respectively. NeuroCheck uses only the first elements of the two arrays, up to the number of elements stated in `uiNumOfInputData` resp. `uiNumOfOutputData`. The information in these arrays is used for the Input Data Configuration dialog and for passing pointers to dynamic data objects during runtime.

Element	**Description**
`piTypeOfInputData`	Pointer to be assigned with static array stating the types of expected input data objects.
`piTypeOfOutputData`	Pointer to be assigned with static array stating the types of created output data objects.

The following symbolic constants are declared in `pi_types.h` for identifiying the possible types of data objects:

Constant	Description
`PI_IMAGE`	Data object is an image.
`PI_HISTO`	Data object is a histogram.
`PI_LAYER`	Data object is a layer of regions of interest.
`PI_MEASARRAY`	Data object is a measurement array.

In addition, a plug-in check function can also use data objects of user-defined types registered in its plug-in DLL.

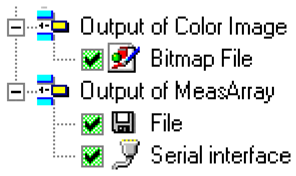
2.6.3.1 Example

The following assignments describe the input / output structure of a function using a gray level image, a histogram and a layer of regions to create a measurement array and a user-definined data type registered with ID `0x1003`:

```
uiNumOfInputData = 3;
static int InputData[]     = { PI_IMAGE,
                               PI_HISTO,
                               PI_LAYER };
piTypeOfInputData = InputData;
uiNumOfOutputData = 2;
static int OutputData[]    = { PI_MEASARRAY,
                               0x1003 };
piTypeOfOutputData = OutputData;
```

2.6.4 Data Output Description

The pointer area of the function info block contains two more pointers. In case of `uiDataOutput` equal to 1, both pointers must be assigned to static arrays with at least `uiNumOfOutputData` elements, each stating if the respective output data objects can take part in the standard NeuroCheck data output to file or to serial interface, respectively. NeuroCheck uses only the first elements of the two arrays, up to the number of elements stated in `uiNumOfOutputData`. The information in these arrays is used for providing output symbols on the Data Output View in manual mode. If data output has been configured for the check routine, NeuroCheck will output the indicated data objects during automatic mode. For objects of a user-defined data type, NeuroCheck will call the dump routine of the respective type whereas output of objects of a standard NeuroCheck data type (image, layer of regions, histogram, measurement list) will be done automatically.



Element	Description
`piMaskFileOutput`	Pointer to be assigned with static array containing flags for the created output data objects for output to file.

`piMaskRs232Output`	Pointer to be assigned with static array containing flags for the created output data objects for output to RS232 serial interface.

The arrays can be seen as mask for enabling the output of output data objects. If element of index `i` is set to `0` (or `FALSE`), there will be no output for the data object with index `i`. If the element is set to `1` (or `TRUE`), output of the respective object is enabled. NeuroCheck will offer the necessary icons on the Data Output View in manual mode and write the required data to file or RS232 serial interface if

- output to the respective destination has been activated in Data Output View globally for the whole check routine,
- output to the respective destination has been activated for the instance of the plug-in check function locally,
- the plug-in check function has been executed successfully during automatic mode

Please note, that output of images (type `PI_IMAGE`) is treated differently to all other data types. Output of images to serial interface is not possible, i.e. enabling an output object of type `PI_IMAGE` in mask `piMaskRs232Output` will be ignored. Output of images to file will not append the image data to the data file configured in NeuroCheck. Instead, NeuroCheck will output the images to bitmap files with sequentially numbered names. Please refer to the **NeuroCheck User Manual** for a detailed description of the output capabilities of NeuroCheck.

2.6.4.1 Example

The following assignments enable the output to file of the first data output object created by the plug-in check function, and output to both file and serial interface of the second one for the output data object description specified in Example 2.6.3.1.

```
uiDataOutput = 1;
static int FileOutput[]    = { TRUE,
                               TRUE };
static int Rs232Output[]   = { FALSE,
                               TRUE };
piMaskFileOutput      = FileOutput;
piMaskRs232Output     = Rs232Output;
```

2.6.5 Descriptive Texts

The string area of the function info block contains four pointers to user-defined strings used for display in the Select New Check Function dialog box. The pointers have to refer to static data elements in the DLL. The strings implement the same level of information in the dialog as for built-in check functions. A recommended maximum size is given although the length of the strings is not limited in principle.

Element	Description
`lpszFunctionName`	Name of function (used in the Select New Check Function dialog, the check routine tree view, protocol files etc.). Recommended string length: max. 40 chars.

`lpszDescription`	Descriptive text for the function.
`lpszHintPos`	Hint displayed in the Select New Check Function dialog when the function can be inserted into the check routine at the current position. Recommended string length: max. 80 chars.
`lpszHintNeg`	Hint displayed in the Select New Check Function dialog when the function cannot be inserted into the check routine at the current position (due to lack of required input data objects). Recommended string length: max. 80 chars.
`alpszCustomViewNames`	List of names for the custom result views of the plug-in check function which will be displayed in the combo box of the Result View pane of NeuroCheck in manual mode. The list must contain `uiNumOfViews` strings. Recommended string length: max. 40 chars.
`alpszInputViewNames, alpszOutputViewNames`	List of names for the standard result views for input and output data objects of the plug-in check function which will be displayed in the combo box of the Result View pane of NeuroCheck in manual mode. The lists must contain `uiNumOfInputData` resp. `uiNumOfOutputData` strings If not specified, predefined names like Gray level image (In) are used. Recommended string length: max. 40 chars.

2.6.6 Function Pointers

The function pointer area of the function info block contains the following elements:

Element	**Description**
`pfnInitialize`	Pointer to initialization routine.
`pfnExecute`	Pointer to execution routine.
`pfnUnInitialize`	Pointer to deinitialization routine.
`pfnParameterDlg`	Pointer to parameter dialog routine.
`pfnViewCreate`	Pointer to view create routine.
`pfnViewDestroy`	Pointer to view destroy routine.

The pointers have been defined using function types declared in `pi_types.h`. Since the individual routine has to adhere to the conventions described in section „Plug-in Check Functions“ function addresses can be assigned directly.
Apart from the parameter dialog routine and the two visualization routines all routines have to

exist for every plug-in check function.
If the parameter dialog routine is missing `pfnParameterDlg` has to be assigned a `NULL` pointer. If the custom visualization is missing `pfnViewCreate` and `pfnViewDestroy` have to be assigned a `NULL` pointer and `uiNumOfViews` has to be assigned a value of 0.

2.7 Structure of a Plug-In Data Type

This section describes the possible capabilities of plug-in data types and the declarations of the routines related to it. A plug-in data type is implemented through two distinct routines:
1. Data output routine.
2. Destroy routine.

The names of these functions are unimportant, because NeuroCheck calls them through function pointers in the type description structure. The declarations though are fixed. All routines must exist for every plug-in data type.

2.7.1 Capabilities of Plug-In Data Types

Plug-in data types can be used for passing arbitrary data between plug-in check functions of the same DLL. Each data object can be of (in principle) unlimited size. NeuroCheck manages this data object as pointer. Only the DLL routines themselves can change the data object so that the programmer of a plug-in DLL can rely on NeuroCheck never to interfere with the inner workings of the object.
Plug-in data types offer a convenient way to pass on arbitrary data to another plug-in check function. Using the view routines of the plug-in check function, this data can be visualized in NeuroCheck. The data output routine enables output of the data to file or RS232 serial interface in the same way as the standard NeuroCheck data types (with the exception of `PI_IMAGE`). Of course, the plug-in programmer may also implement a custom output in the data output routine.

2.7.2 Data Output Routine

The data output routine of a plug-in data type has to be declared as follows (excepting the actual name of course):

```
BOOL WINAPI DataOutput(sPI_TARGET_INFO* const psTargetInfo,
                       void* const pData,
                       sPI_DATAOUTPUT_INFO* psDataContainer);
```

In `psTargetInfo` NeuroCheck passes information about the output target for which the data output routine is called. The following structure is used for the target description:

```
typedef struct
{
    unsigned int      uiType;
    LPSTR             lspzName;
}   sPI_TARGET_INFO;
```

The structure elements have the following meaning:

Element	Description
`uiType`	Type of target. Can have one of two values, declared as symbolic

	constants in `pi_types.h`: `PI_TARGET_FILE`: Output to data file. `PI_TARGET_RS232`: Output to RS232 serial interface.
`lspzName`	Name of target as a zero-terminated string. For output to data file, the string contains the full name of the output file. For output to RS232 serial interface, the string contains the name of the COM port.

All elements of the structure have to be considered „read-only“.
In `pData` NeuroCheck passes the address of the data object allocated previously inside the execution routine. It is passed as *constant pointer* which means that the pointer itself must not be changed. In `psDataContainer` the address of an allocated data container is passed to the routine for returning the data sequence that NeuroCheck will write to the given output target. The following structure is used for the data container:

```
typedef struct
{
    unsigend int            uiCount;
    sPI_DATAOUTPUT_ITEM*    psDataItem;
}
    sPI_DATAOUTPUT_INFO;
```

The structure elements have the following meaning:

Element	**Description**
`uiCount`	Number of data items, i.e. length of array `psDataItem`.
`psDataItem`	Array of data items. The array must be allocated by the plug-in DLL using `VirtualAlloc`. De-allocation will be handled by NeuroCheck.

Each data item can be either an integer value, a floating-point value, or a string value. The following structure is used for the data items:

```
typedef struct
{
    unsigned int      uiType;
    int               iData;
    float             fData;
    char*             pszData;
}
    sPI_DATAOUTPUT_ITEM;
```

The structure elements have the following meaning:

Element	**Description**
`uiType`	Type of data item. Can have one of three values, declared as symbolic constants in `pi_types.h`: `PI_ITEM_INT` item is integer value in `iData`. `PI_ITEM_FLOAT` item is floating point value in `fData`. `PI_ITEM_STRING` item is string value in `pszData`.
`iData`	Integer value of data item.

`fData`	Floating point value of data item.
`pszData`	String value of data item. The string must be allocated by the DLL using `VirtualAlloc` `pszData`. Please note that `pszData` must be assigned to `NULL` if `iType` is not `PI_ITEM_STRING`.

2.7.2.1 Example

The following sample code shows how to output the three different types of data (Int, Float and String), assuming that the plug-in data object was defined as structure `sMY_DATA_TYPE`:

```
typedef struct
{
    int   iData;      // integer value
    float fData;      // float value
} sMY_DATA_TYPE;

BOOL WINAPI DataOutput(    sPI_TARGET_INFO* const psTargetInfo,
                           void* const pData,
                           sPI_DATAOUTPUT_INFO* psDataOutputInfo)
{

    // verify data pointer
    if (pData == NULL) return FALSE;

    // type cast of data pointer
    sMY_DATA_TYPE* psMyData = (sMY_DATA_TYPE *) pData;

    // output of 3 elements: int, float and string
    psDataOutputInfo->uiCount = 3;

    // allocate memory for the elements (using the v-heap api!)
    psDataOutputInfo->psDataItem = (sPI_DATAOUTPUT_ITEM*) VirtualAlloc(
                              NULL,
                              (DWORD)(psDataOutputInfo->uiCount *
                                           sizeof(sPI_DATAOUTPUT_ITEM)),
                              MEM_COMMIT,
                              PAGE_READWRITE);

    // insert integer element
    psDataOutputInfo->psDataItem[0].uiType = PI_ITEM_INT;
    psDataOutputInfo->psDataItem[0].iData = psMyData->iIntegerValue;

    // insert float element
    psDataOutputInfo->psDataItem[1].uiType = PI_ITEM_FLOAT;
    psDataOutputInfo->psDataItem[1].fData = psMyData->fFloatValue;

    // get element for string output
    sPI_DATAOUTPUT_ITEM* psItem = &psDataOutputInfo->psDataItem[2];

    // insert string element
    psItem->uiType = PI_ITEM_STRING;

    // create string for output
    CString csOutputString = "Example for output of a string!";

    // get string length
```

```
        int iStrLen = (int) csOutputString.GetLength();

        // zero terminated string!
        iStrLen += 1;

        // allocate memory for string (using the v-heap api!)
        psItem->pszData =
            (char *) VirtualAlloc(NULL,
                                  (DWORD)(iStrLen * sizeof(char)),
                                  MEM_COMMIT,
                                  PAGE_READWRITE);

        // copy string
        strcpy(psItem->pszData, csOutputString.GetBuffer(iStrLen));

        return TRUE;

    } // DataOutput
```

2.7.3 Destroy Routine

The destroy routine of a plug-in check function has to be declared as follows (excepting the actual name of course):

```
void WINAPI TypeDestroy(void* pbyData);
```

In `pbyData` NeuroCheck passes the address of the data object allocated previously inside the execution routine. The destroy routine is called whenever a data object of this plug-in data type is to be de-allocated. Usually this will be before the next execution of the single check containing the plug-in check function that created such an object. Inside the destroy routine, the data passed in `pbyData` should be de-allocated using an operator that matches the operator previously used for allocation.

2.8 The Type Description Block

Function `PI_GetTypeDesc(uiIndex, psFctDesc)` has to fill the type description block whose address is passed in `psFctDesc` for every possible index `uiIndex` with the characteristics of the respective plug-in data type. This section describes the structure of the type description block and the meaning of its members.

2.8.1 Type Description Block Declaration

The type description block is declared as follows (detailed explanations of the individual sections follow below):

```
typedef struct
{
    // version control
    unsigned int                uiStructSize;

    // data
    unsigned int                uiTypeId;
    int                         iIconIndex;

    // - strings
    LPSTR                       lpszTypeDesc;

    // - function pointer
    PFNPlugInTypeDataOutput     pfnDataOutput;
```

```
        PFNPlugInTypeDestroy            pfnTypeDestroy;
    }
        sPI_TYPE_DESC;
```

2.8.2 General Information

The data area of the type description block contains the following elements:

Element	Description
`uiStructSize`	NeuroCheck allocates the block and initializes each element to 0 with the exception of its first element `uiStructSize`. This element contains the size of the type description block as returned by the `sizeof()` function. The value of `uiStructSize` should be considered read-only and can be used inside the plug-in DLL to verify that NeuroCheck and the plug-in DLL use the same declaration of the `sPI_TYPE_DESC` structure. If the value of `uiStructSize` and the return value of `sizeof(sPI_TYPE_DESC)` are not equal, there is a version conflict between the plug-in interface provided by NeuroCheck and the one expected by the DLL.
`uiTypeId`	Unique identification number for the plug-in data type. NeuroCheck uses this number for various purposes, among them identifying data types used by a plug-in check function. For this reason plug-in data types inside a single plug-in DLLs must not use identical type IDs. **NOTE:** type IDs of plug-in data types must not interfere with the type IDs of NeuroCheck data types. Therefore the IDs must be within a range limited by symbolic constants `DATATYPE_ID_MIN` and `DATATYPE_ID_MAX` declared in `pi_types.h`. All values outside this range are reserverd for use by NeuroCheck GmbH and partners. The symbolic constants currently are defined to `0x1000` and `0x2000`, which leaves 4096 entries for custom use.
`iIconIndex`	Identification number of icon that will be used by NeuroCheck for the plug-in data type. If `iIconIndex` contains a negative value, then NeuroCheck will display a default icon for the plug-in data type.
`lpszTypeDesc`	Descriptive text for the plug-in data type. The pointers has to refer to a static data element in the DLL. The length of the string must not exceed 40 characters.

2.8.3 Function Pointers

The function pointer area of the type description block contains the following elements:

Element	Description
`pfnDataOutput`	Pointer to data output routine.
`pfnTypeDestroy`	Pointer to destroy routine of plug-in data type.

The pointers have been defined using function types declared in `pi_types.h`. Since the individual routine has to adhere to the conventions described in section „Plug-in Check Functions“ function addresses can be assigned directly.
Both the data output and the destroy routine have to exist for every plug-in data type.

3 Data Types and Definitions

This chapter describes the data types and definitions for the use of dynamic data objects declared in the included header file `pi_types.h`. Please note that in several cases different rules apply for access to an input data objects and for creating an output data object of the same data type.

3.1 Image

The following structure is used by NeuroCheck for images:

```
typedef struct
{
    unsigned int      uiWidth;
    unsigned int      uiHeight;
    BOOL              bColor;
    BYTE*             pbyGrayValue;
    BYTE*             pbyRedValue;
    BYTE*             pbyGreenValue;
    BYTE*             pbyBlueValue;
    int               iSource;
    LPCTSTR           lpszSourceName;
}   sPI_IMAGE;
```

The structure elements have the following meaning:

Element	Description
`uiWidth`	Width of image in pixels.
`uiHeight`	Height of image in pixels.
`bColor`	`TRUE` for color image, `FALSE` for gray level image.
`pbyGrayValue`	Pointer to array with gray level values.
`pbyRedValue`	Pointer to array with values of red channel for color image, `NULL` for gray level image.
`pbyGreenValue`	Pointer to array with values of green channel for color image, `NULL` for gray level image.
`pbyBlueValue`	Pointer to array with values of blue channel for color image, `NULL` for gray level image.
`iSource`	Source of image. Can have one of the following values, declared as symbolic constants in `pi_types.h`: `PI_IMSRC_FILE` bitmap file. `PI_IMSRC_CAM` camera. `PI_IMSRC_TRAY` image tray. `PI_IMSRC_UNKNOWN` cannot be determined, e.g. after an addition of images.

`lpszSourceName`	Description of image source as a zero-terminated string, i.e. name of bitmap file or name of camera. The contents of this string must not be changed by a plug-in check function

Each array with pixel values (gray level or color) represents the pixel matrix of the image and thus must have a size of `uiWidth` times `uiHeight`. The pixels are ordered according to the NeuroCheck coordinate system which starts at origin (0,0) in the top left corner with its positive X axis pointing to the right and its positive Y axis pointing downwards. In the array representation, the lines of the image are simply appended.
For an input image object, a plug-in check function may only alter the pixel values of the gray level array and, for color images only, of the three color channels. All other elements should be considered „read-only".
Allocating a new image object, all elements except `iSource` and `lpszSourceName` can be assigned. Each image must have a valid gray level array of the size specified by `uiWidth` and `uiHeight`. The array must be allocated with `VirtualAlloc`. In addition, each color image must have a valid array for each color channel. Please note that for a color image the gray level array always should be filled with valid values, too. For color images created by NeuroCheck, the gray level array simply contains the same values as the green channel by default.

3.2 Histogram

The following structure is used by NeuroCheck for gray level histograms with binarization thresholds:

```
typedef struct
{
    unsigned short int    uiThreshold;
    unsigned int          auiHistoBuffer[256];
}   sPI_HISTO;
```

The structure elements have the following meaning:

Element	Description
`uiThreshold`	Binarization threshold
`auiHistoBuffer`	Array with number of pixels for each gray level.

Both for input and output objects, the value of `uiThreshold` and the values of `auiHistoBuffer` can be modified or assigned.

3.3 Regions of Interest

In NeuroCheck regions of interest (ROIs) can be manually defined or dynamically created, i.e. segmented from image objects. Regions are organized within layers which can contain both types of objects. The description of this complex data structure is subdivided into the following sections:

1. Layer of regions of interest.
2. Single region.
3. Object contour.
4. Object region.
5. Model geometries.
6. Creating a new region.

Please note, that where not indicated differently, for all coordinates the standard NeuroCheck coordinate system applies which starts with its origin (0,0) in the top left corner of the image with its positive X axis pointing to the right and its positive Y axis pointing downwards.

3.3.1 Layer of Regions of Interest

Every function creating regions of interest, be it manually or automatically through an object search, generates exactly one layer of regions with the following structure:

```
typedef struct
{
    unsigned int      uiCount;
    BOOL              bMeasurement[300];
    LPSTR             lpszMeasurementName[300];
    unsigned int      uiClasses;
    LPSTR*            alpszClasses;
    sPI_OBJECT*       pasObject;
}   sPI_LAYER;
```

The structure elements have the following meaning:

Element	Description
`uiCount`	Number of objects in layer.
`bMeasurement`	Array of flags for the activation of individual feature values. The various features are identified by symbolic constants defined in `pi_types.h`.
`lpszMeasurementName`	Array of string pointers holding the names of the features activated by `bMeasurement`. For an input list, these strings are read-only. Creating a new output list, custom feature names can be specified by assigning pointers to static strings.
`uiClasses`	Number of classes available for the layer if its objects have been classified. The class ID assigned to each object is given as feature value. Class information is created by check functions Classify ROIs, Template Matching, and Color Matching.
`alpszClasses`	Array of class names. Element `[i]` of the array represents the name of class ID `i`. These strings are read-only.
`pasObject`	Pointer to array of region descriptions.

NOTE: the only operations a plug-in check function may perform within this structure are setting flags in the `bMeasurement` array to activate or deactivate features, and to specify custom feature names when creating a new output list. The number of regions and the addresses of the individual regions must not be changed! Thus a plug-in check function cannot delete a region (but it can declare the object invalid so that a subsequent call to function Screen ROIs removes the region. See section „Single Region of Interest" for details). Likewise a plug-in check function cannot add regions to a layer, but it can create a new layer, copy regions to this layer and add additional objects internally.

3.3.2 Single Region of Interest

NeuroCheck uses the following structure to describe a single region of interest (ROI):

```
typedef struct
{
    /* ------* Read only *------ */
    int                 iType;
    int                 iWidth;
    int                 iHeight;
    sPI_CONTOUR*        psContour;
    sPI_REGION*         psRegion;
    // model geometries
    int                 iFitType;
    float               fFitParameters[10];

    /* ------------------------------------ */
    /* ------* Modify/Write *------ */
    int                 iNumber;
    int                 iGroupNumber;
    BOOL                bValid;
    int                 iX;
    int                 iY;
    float               fMeasurement[300];
    /* -------* Write only *------ */
    int                 iShapeType;
    int                 iSearchReg;
    sPI_POINTS*         psPoints;
}   sPI_OBJECT;
```

The structure elements have the following meaning:

Element	Description
`iType`	Type of ROI. Can have one of three values, declared as symbolic constants in `pi_types.h`: `PI_OBJECTAOI`: ROI contains an enclosing rectangle only. `PI_OBJECTCONTOUR`: ROI also contains a valid contour, but no region. `PI_OBJECTREGION`: ROI also contains a valid contour and a valid region.
`iWidth, iHeight`	Width and height of enclosing rectangle.
`psContour`	Description of the object contour.

`psRegion`	Description of the object region.
`iFitType`	Description of a model geometry calculated for the ROI. The model geometry can be identified by symbolic constants defined in `pi_types.h`.
`fFitParameters`	Array with parameters of the calculated model geometry. See „Model Geometries“.
`iNumber`	Unique number of region inside a group.
`iGroupNumber`	Group number of region.
`bValid`	Validity of object, intended for user defined screening function. These must not delete a region but declare it invalid by setting this element to `FALSE`. NeuroCheck will remove all invalid regions when the layer structure is read back to the internal NeuroCheck format.
`iX, iY`	Coordinates of the top left corner of the enclosing rectangle for input objects, coordinates of reference point for output objects (see „Creating a new Region of Interest.“).
`fMeasurement`	Feature array. Only array elements for which the corresponding element in the `bMeasurement` array of the layer structure is `TRUE` contain valid information. Features can be identified by symbolic constants defined in `pi_types.h`.
`iShapeType`	Type of new ROI, declared as symbolic constants in `pi_types.h`
`iSearchReg`	Size of surrounding area in pixels for new ROI.
`psPoints`	Description of new ROI for output object, NULL for input object.

The elements of the structure are seperated into three categories:

- „read-only“ elements
- „modify/write“ elements
- „write-only“ elements

„Read-only“ elements must not be changed for input objects nor be assigned for output objects. For input objects, only „modify/write“ can be changed. Creating a new output data object, the plug-in check function has to assign the „modify/write“ and „write-only“ elements in the structure.
A plug-in check function altering „modify/write“ elements in the structure for an input object has following effects:

Modify element	Purpose
`iNumber`	Changes sorting of regions.

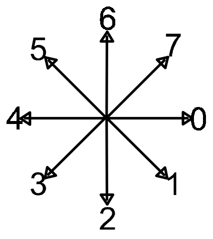
`iGroupNumber`	Changes group membership of regions.
`bValid`	Assigning `FALSE` lets region be removed.
`iX, iY`	Changes position of region.
`fMeasurement[i]`	Changes value of feature `[i]`.

3.3.3 Object Contour

Polylines and ROIs created dynamically by object segmentation usually have an object contour described by the following structure:

```
typedef struct
{
    int      iXStart;
    int      iYStart;
    int      iLength;
    BYTE *   pbyChain;
}   sPI_CONTOUR;
```

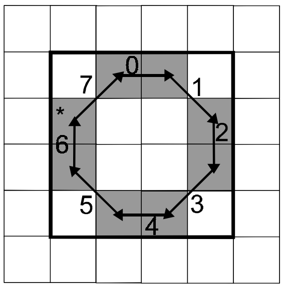
The structure elements have the following meaning:

Element	Description
`iXStart, iYStart`	Coordinates of the first contour point, relative to enclosing rectangle.
`iLength`	Number of contour points.
`pByChain`	Chain code of the contour. The elements of the chaing code indicate the direction of the contour from the current point to the next point. The following figure visualizes the possible values: 

A plug-in check function must not change any values in this data structure, all elements are to be considered „read-only".

Example for Contour Description

The figure shows a dark octogon on a light background and the corresponding contour description. The description of the enclosing rectangle is given by X=1, Y=1 and both width and heigth equal to 4. The asterisk marks the start of the contour, the numbers next to each section of the contour encode the direction of the connection between the two contour points.



```
iXStart = 0;
iYStart = 1;
iLength = 8;
pByChain → [7, 0, 1, 2, 3, 4, 5, 6]
```

3.3.4 Object Region

ROIs created dynamically by object segmentation usually have an object region described by the following structure:

```
typedef struct
{
    int         iLength;
    int*        piX;
    int*        piY;
    int*        piXLength;
}   sPI_REGION;
```

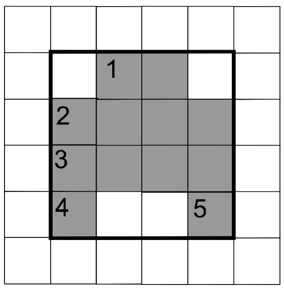
The structure elements have the following meaning:

Element	Description
`iLength`	Length of run length code i.e. number of line segments.
`piX, piY`	Arrays containing the X and Y coordinates of the starting positions for each RLC line segment relative to enclosing rectangle. Both pointers must be assigned to arrays of length `iLength`.
`piXLength`	Array containing the length in X direction of each line segment. The pointer must be assigned to an array of length `iLength`.

A plug-in check function must not change any values in this data structure, all elements are to be considered „read-only".

Example for Region Description

The figure shows a dark octogon on a light background and the corresponding region description. The description of the enclosing rectangle is given by X=1, Y=1 and both width and heigth equal to 4. The numbers mark the start of each line segment and indicate the ordering. For instance, the first line segment starts at the relative position `X=1`, `Y=0` (absolute position X=2, Y=1) and has a length of 2 pixels.



```
iLength = 5;
piX       → [1, 0, 0, 0, 3]
piY       → [0, 1, 2, 3, 3]
piXLength → [2, 4, 4, 1, 1]
```

3.3.5 Model Geometries

Each ROI can be assigned a model geometry as calculated by check function Compute Model Geometries. The type of the model geometry can be identified by the value of element `iFitType`. It can have one of four values, declared as symbolic constants in `pi_types.h`:

Constant	Meaning
`PI_FIT_CONTOUR`	No specific model geometry calculated.
`PI_FIT_POINT`	Model geometry is a single point.
`PI_FIT_LINE`	Model geometry is a line.
`PI_FIT_CIRCLE`	Model geometry is a circle.

Array `fFitParameters` contains the parameters describing the specified model geometry. The meaning of the elements of array are listed in following table:

Model geometry	Meaning of elements of **`fFitParameters`**
`PI_FIT_CONTOUR`	If no specific model geometry is calculated, array `fFitParameters` contains the coordinates of the center of gravity of the object contour: `fFitParameters[0]` X coordinate of center point `fFitParameters[1]` Y coordinate of center point
`PI_FIT_POINT`	The description of a point simply consists of its coordinates: `fFitParameters[0]` X coordinate of point `fFitParameters[1]` Y coordinate of point
`PI_FIT_LINE`	The description of a line is given by the coordinates of its center point and the parameters of its mathematical description, i.e. slope parameter `m` and constant `b`. Furthermore, the two points of intersection with the search area are given which represent the start and end point of the line.

	`fFitParameters[0]`	X coordinate of center point
	`fFitParameters[1]`	Y coordinate of center point
	`fFitParameters[2]`	Slope parameter `m`
	`fFitParameters[3]`	Y axis parameter `b`
	`fFitParameters[4]`	X coordinate of start point
	`fFitParameters[5]`	Y coordinate of start point
	`fFitParameters[6]`	X coordinate of end point
	`fFitParameters[7]`	Y coordinate of end point
`PI_FIT_CIRCLE`	The description of a circle is given by the coordinates of its center point and its radius:	
	`fFitParameters[0]`	X coordinate of center point
	`fFitParameters[1]`	Y coordinate of center point
	`fFitParameters[2]`	Radius in pixels

A plug-in check function must not change any values in `fFitParameters`, all elements are to be considered „read-only".

3.3.6 Creating a new Region of Interest.

Creating a new ROI, the modify and write-only elements of structure `sPI_OBJECT` must be assigned. For simplicity, the plug-in check function creating the new ROI does not have to calculate the chain code of its contour or the run length code of its region. Instead, the shape of the new region is specified by a number of points described by the following structure:

```
typedef struct
{
    int      iLength;
    int*     piX;
    int*     piY;
}   sPI_POINTS;
```

The structure elements have the following meaning:

Element	Description
`iLength`	Length of arrays, i.e. number of points.
`piX, piY`	Arrays containing the X and Y coordinates of the points. Both pointers must be assigned to arrays of length `iLength`. Please note that the coordinates are relative to the reference point given by parameters `iX` and `iY` in structure `sPI_OBJECT`.

The interpretation of the given points depends on the value of `iShapeType`. It can be assigned one of following values, declared as symbolic constants in `pi_types.h`:

Constant	Meaning
`PI_SHAPE_AOI`	Rectangular AOI. The enclosing rectangle for the given points is calculated. At least two points are required.
`PI_SHAPE_POLY`	Polyline. The polyline is defined by drawing lines from point one to point two, from point two to point three, etc. The polyline will

be a closed polyline, if the last point is identical to the first one. At least two points are required.

`PI_SHAPE_POLY_REGION`
Closed polyline for which filling is activated. Note that if the polyline is not closed, an additional line will automatically be inserted from the first point to the last one. At least four points are required, the last one identical to the first one.

`PI_SHAPE_POLY_SEARCHREG`
Polyline with a surrounding area. The size of the surrounding area is given by the value of element `iSearchReg`. Note that the surrounding area must not leave the borders of the applied image.

`PI_SHAPE_POLY_SEARCHREG_REGION`
Polyline with a surrounding area for which filling is activated. The size of the surrounding area is given by the value of element `iSearchReg`. At least two points are required

`PI_SHAPE_CIRCLE`
Polyline for which circle conversion is activated. Circle if the polyline is closed, otherwise arc. At least three points are required.

`PI_SHAPE_CIRCLE_REGION`
Closed polyline for which both circle conversion and filling is activated. Note that if the polyline is not closed, an additional line will automatically be inserted from the first point to the last one. At least four points are required, the last one identical to the first one.

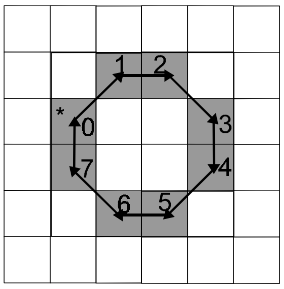
`PI_SHAPE_CIRCLE_SEARCHREG`
Polyline with a surrounding area for which circle conversion is activated. The size of the surrounding area is given by the value of element `iSearchReg`. At least three points are required.

`PI_SHAPE_CIRCLE_SEARCHREG_REGION`
Polyline with a surrounding area for which circle conversion and filling is activated. The size of the surrounding area is given by the value of element `iSearchReg`. At least three points are required.

Note that the ordering of the points is crutial for any polyline interpretation. There should be no crossing of polylines. The created region must not pass the edge of the image section it will be applied to. An arbitrary object contour can be approximated by a closed polyline by specifying every single point of the contour.

Example for Creating a Contour

The figure shows a dark octogon on a light background and the corresponding description of a closed polyline for approximating its contour. The asterisk marks the start of the contour, the numbers the point numbers, and the arrows the polylines to the next contour point. The top left corner (`X = 0` and `Y = 0`) has been chosen as reference point.



`iShapeType = PI_SHAPE_POLY;`

`iSearchReg = 0;`

Elements of `psPoints`:

`iLength = 8;`

piX → `[1, 2, 3, 4, 4, 3, 2, 1, 1],`

piY → `[2, 1, 1, 2, 3, 4, 4, 3, 2],`

3.4 Measurement Array

Measurement arrays are used within NeuroCheck for geometrical measurements created by function Gauge ROIs. This function is capable of measuring relations between different regions. Therefore its results cannot be assigned to individual regions and require a new data structure:

```
typedef struct
{
    unsigned int      uiCount;
    sPI_MEASVALUE*    pasMeasValue;
}   sPI_MEASARRAY;
```

The structure elements have the following meaning:

Element	Description
`uiCount`	Number of values in measurement array.
`pasMeasValue`	Array of measurements.

A plug-in check function must not change anything in this structure. All elements are to be considered „read-only“.

Individual measurements are described by the following structure:

```
typedef struct
{
    unsigned int uiNumber;
    float        fMeasValue;
    LPSTR        lpszName;
}   sPI_MEASVALUE;
```

The structure elements designate:

Element	Description
`uiNumber`	Identification number of the measurement.
`fMeasValue`	Measurement value. Can be modified for input objects.
`lpszName`	Description of measurement as a zero-terminated string (this description is entered by the user). For input data objects, the contents of this string must not be changed by a plug-in check function. Creating a new output data object, the plug-in check function may assign the address of a static string inside the DLL to the pointer, just like it is done with the check function name and description strings. In this case, the maximum length of the string must not exceed 80 characters. However, for readability we recommend a maximum length of 40 characters.

3.5 Symbolic Constants

3.5.1 Identification of Data Objects

There are only three symbolic constants in this section representing the possible values of element `iType` in the structure `sPI_OBJECT` describing a region. They allow to distinguish between objects with and without a complete contour description.

Constant	Description
`PI_OBJECTAOI`	Region contains enclosing rectangle only
`PI_OBJECTCONTOUR`	Region also contains contour description.
`PI_OBJECTREGION`	Region also contains contour and region description.

3.5.2 Feature Values

These constants serve to distinguish between the individual feature values possible for a region. They are intended to be used as indices for the `fMeasurement` array in the region description structure `sPI_OBJECT` and the `bMeasurement` and `lpszMeasurementName` arrays in the region layer structure `sPI_LAYER`.
The constants are listed here in the same order as the features in the NeuroCheck documentation which does not necessarily correspond to the order in `pi_types.h`. To ensure compatibility to future versions of the plug-in interface programmers should always use the symbolic names, never the values directly. The features themselves are explained in detail in the NeuroCheck documentation.

Center of Gravity

Constant	Description
`PI_ALL_XCOFG`	X coordinate of the center of gravity of the region.
`PI_ALL_YCOFG`	Y coordinate of the center of gravity of the region.

Enclosing Rectangle

Constant	Description
`PI_AOI_XORG`	X coordinate of top left corner of the enclosing rectangle.
`PI_AOI_YORG`	Y coordinate of top left corner of the enclosing rectangle.
`PI_AOI_XDIM`	Width of enclosing rectangle.
`PI_AOI_YDIM`	Height of enclosing rectangle.
`PI_AOI_RATIO`	Ratio of height to width of enclosing rectangle.

Axes

Constant	Description
`PI_BNDAOI_LAXIS`	Length of principal axis of the region.
`PI_BNDAOI_SAXIS`	Length of secondary axis of the region.
`PI_BNDAOI_LANGLE_180`	Direction of principal axis without regard to orientation (i.e. between 0 and 180°)
`PI_BNDAOI_LANGLE_360`	Direction of principal axis with regard to orientation (i.e. between 0 and 360°)

Radii

In NeuroCheck a radius is defined as the distance from the center of gravity to a point on the border of the region.

Constant	Description
`PI_BNDAOI_RADMEAN`	Average radius.
`PI_BNDAOI_RADMIN`	Minimum radius.
`PI_BNDAOI_RADMAX`	Maximum radius.
`PI_BNDAOI_RANGLE`	Angle between minimum and maximum radius.

General Shape Description

Constant	Description
`PI_ALL_AREA`	Area of region.
`PI_BNDAOI_PERI`	Circumference of region.
`PI_BNDAOI_COMPACT`	Form factor of region (defined as Area/(4 * π * Circumference²); maximum value 1 for ideal circles, else smaller).
`PI_BNDAOI_FIBRELENGTH`	Approximate length of a line following the shape of the region holding equal distance to both edges.
`PI_BNDAOI_FIBREWIDTH`	Sum of distances from the fibre to both edges.
`PI_BNDAOI_ELONGATION`	Ratio of fibre length to width.

Border Contact

Constant	Description
`PI_AOI_ANY`	Region touches any image border.
`PI_AOI_UP`	Region touches top border of image.
`PI_AOI_DOWN`	Region touches bottom border of image.
`PI_AOI_LEFT`	Region touches left border of image.
`PI_AOI_RIGHT`	Region touches right border of image.

Number of Enclosed Objects

Constant	Description
`PI_AOI_HOLES`	Number of holes enclosed in the region.

Gray Level Statistics

Constant	Description
`PI_ALL_MEAN`	Average gray level inside region.
`PI_ALL_MIN`	Minimum gray level inside region.
`PI_ALL_MAX`	Maximum gray level inside region.
`PI_ALL_SIGMA`	Standard deviation of gray levels inside region.

`PI_ALL_CONTRAST`	Maximum difference of gray levels inside region.

Gradient Statistics

Constant	Description
`PI_ALL_GRADMEAN`	Average gradient inside region.
`PI_ALL_GRADMAX`	Maximum gradient inside region.
`PI_ALL_GRADSIGMA`	Standard deviation of gray levels inside region.

Curvature Statistics

Constant	Description
`PI_GEN_CURV_MEAN`	Average curvature.
`PI_GEN_CURV_SIGMA`	Standard deviation of curvature values.
`PI_GEN_CURV_MIN`	Minimum curvature.
`PI_GEN_CURV_MAX`	Maximum curvature.
`PI_GEN_CURV_CONTRAST`	Maximum amplitude of curvatures along the contour.

Results of Determine Position

Constant	Description
`PI_POS_OFFSET_X`	Offset in X direction calculated by Determine Position.
`PI_POS_OFFSET_Y`	Offset in Y direction calculated by Determine Position.
`PI_POS_ROT_ANGLE`	Rotation angle calculated by Determine Position.
`PI_POS_PIVOT_X`	Pivot X for Determine Position.
`PI_POS_PIVOT_Y`	Pivot Y for Determine Position.

Results of Template Matching

Constant	Description
`PI_GEN_COR_QUALITY`	Quality (correlation coefficient of region with its template).
`PI_GEN_COR_SUBPIX_X`	X coordinate of region found by subpixel template matching.
`PI_GEN_COR_SUBPIX_Y`	

	Y coordinate of region found by subpixel template matching.
`PI_GEN_COR_ANGLE`	Angle of rotated template.

Results of Classification

Constant	Description
`PI_GEN_CLASS`	Number of class determined by classifier (or any other function which generates class information).
`PI_GEN_CLASSQUALITY`	Quality of classification result (equals the classification certainty in case of the classifier, the correlation coefficient in the case of template matching).

3.5.3 User-Defined Features

Plug-in check functions can compute additional feature values and store them in free positions of the feature array. For this purpose indizes ranging from `60` to `99` have been reserved (since a plug-in DLL is typically used for implementing functionality specific to a certain inspection problem, this range can be used for different purposes in different DLLs so that there should be no shortage of feature indizes).
For a newly created layer of regions of interest, a custom designation of the feature can be given which will be used in the NeuroCheck functions working with this feature. This can be done by assigning a pointer to a static string in the `lpszMeasurementName` array of the `sPI_LAYER` structure.

3.5.4 Target Types

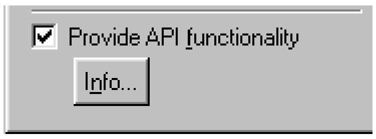
For data output, two target types are supported:

Constant	Description
`PI_TARGET_FILE`	Output to data file. `lpszTarget` contains the full name of the output file.
`PI_TARGET_RS232`	Output to RS232 serial interface. `lpszTarget` contains the name of the COM port.

4 NeuroCheck API Functionality

The NeuroCheck API (application program interface) provides a way to call certain functionality of the NeuroCheck main program from the plug-in DLL. For instance, it enables read and write access to the same digital I/O and field bus devices used in NeuroCheck. Another main feature offers a convenient and simple way to convert pixel arrays into Windows® bitmap format which is necessary for displaying images under Windows®.

The NeuroCheck API functionality is encapsulated in a dynamic link library called `Ncapi.dll`. If NeuroCheck can locate this DLL in its installation path, on page Plug-In of the General Software Settings dialog box appears a check box „Provide API functionality“ and an Info button for displaying a message box with information about `Ncapi.dll`. If the check box is activated, the NeuroCheck API can be used by any plug-in DLL. For using the functionality, the plug-in DLL must connect to DLL `Ncapi.dll` as explained below.



The usage of the API functionality is demonstrated in the plug-in samples `PI_NcApiCalls` and `PI_Visualization` (see 5.1 Overview of the Sample Plug-In DLLs). There you also can find a convenient wrapper class implementation for C++. By inserting the files `PI_NcApi.h` and `PI_NcApi.cpp` into your project, you can access all API functions through methods of the class `CNcApi`. For some API functions, the class also provides overloaded methods which simplify their useage with C++.

The functions provided by the API are devided into the following categories:

- Access to hardware devices, i.e. digital I/O and field bus devices.
- Bitmaps management.
- Access to NeuroCheck settings.
- Error Codes.

4.1 Access to Hardware Devices

For access to the hardware configured in NeuroCheck the API provides a number of functions. The main application will be access to digital I/O and field bus devices.

GetDeviceCount()

```
extern "C" WINAPI int GetDeviceCount(int iDeviceType);
```

The `DeviceCount` function returns the number of devices of the type specified by `iDeviceType` configured in NeuroCheck's Device Manager. If the input argument is incorrect, the function returns -1. For instance, the function can be used to retrieve information about the number of I/O boards configured in NeuroCheck.

Parameters

`iDeviceType`	Type of device. The different device types are identified by symbolic constants defined in `pi_NcApi.h`.

GetDeviceName()

```
extern "C" WINAPI LPSTR GetDeviceName(int iDeviceType, int iDeviceIndex);
```

The `GetDeviceName` function returns the name of the specified device as configured in NeuroCheck's Device Manager. The string is returned as pointer to a static string. This pointer is only valid until the next call to this function, so it is recommended to store the string immediately in a CString variable. It is not possible to change the contents of the string. If one of the input arguments is incorrect, the function returns a NULL pointer.

Parameters

`iDeviceType`	Type of device. The different device types are identified by symbolic constants defined in `pi_NcApi.h`.
`iDeviceIndex`	Index of device.

GetSubDeviceCount()

```
extern "C" WINAPI int GetSubDeviceCount(
                               int iDeviceType,
                               int iDeviceIndex,
                               int iSubDeviceType);
```

The `GetSubDeviceCount` function returns the number of sub devices of the type specified by `iDeviceType`, `iDeviceIndex` and `iSubDeviceType` configured in NeuroCheck's Device Manager. If the input arguments are incorrect, the function returns -1

Parameters

`iDeviceType`	Type of parent device. The different device types are identified by symbolic constants defined in `pi_NcApi.h`.
`iDeviceIndex`	Index of parent device.
`iSubDeviceType`	Type of sub device. The different sub device types are identified by symbolic constants defined in `pi_NcApi.h`.

GetSubDeviceName()

```
extern "C" WINAPI LPSTR GetSubDeviceName(
                               int iDeviceType,
                               int iDeviceIndex,
                               int iSubDeviceType,
                               int iSubDeviceIndex);
```

The `GetSubDeviceName` function returns the name of the specified sub device as configured in NeuroCheck's Device Manager. The string is returned as pointer to a static string. This pointer is only valid until the next call to this function, so it is recommended to store the string immediately in a CString variable. It is not possible to change the contents of the string. If one of the input arguments is incorrect, the function returns a NULL pointer. For instance, the function can be used to read the user-defined names of the digital input bits.

Parameters

`iDeviceType`	Type of parent device. The different device types are identified by symbolic constants defined in `pi_NcApi.h`.
`iDeviceIndex`	Index of parent device.
`iSubDeviceType`	Type of sub device. The different sub device types are identified by symbolic constants defined in `pi_NcApi.h`.
`iSubDeviceIndex`	Index of sub device.

ReadDigitalInput()

```
extern "C" WINAPI BOOL ReadDigitalInput(int iBoardIndex, int iInputNumber);
```

The `ReadDigitalInput` function reads the current state of a digital input. It takes two arguments which specify the digital I/O board and the input to be read. The function returns the state of the input, i.e. `TRUE` or `FALSE`.

Parameters

`iBoardIndex`	Index of digital I/O board.
`iInputNumber`	Number of input to be read.

SetDigitalOutput()

```
extern "C" WINAPI BOOL SetDigitalOutput(
                               int iBoardIndex,
                               int iOutputNumber,
                               BOOL bNewState);
```

The `SetDigitalOutput` function sets the state of a digital output. It takes three arguments specifying the digital I/O board, the output to be set, and the state the output will be set to. The function returns `TRUE` if the output could be set, or `FALSE` if an error occurred.

Parameters

`iBoardIndex`	Index of digital I/O board.

`iOutputNumber`	Number of output to be set.
`bNewState`	New state value.

ReadDigitalInputWord()

```
extern "C" WINAPI BOOL ReadDigitalInputWord(
                               int iBoardIndex,
                               int * pInputWord);
```

The `ReadDigitalInputWord` function reads the current states of all 16 digital inputs of one digital I/O board. It takes two arguments, one which specifies the digital I/O board to be read and one to return an integer value encoding the states of the inputs. The function returns `TRUE` if the input word could be read, or `FALSE` if an error occurred.

Parameters

`iBoardIndex`	Index of digital I/O board.
`pInputWord`	Pointer to integer value, used to return decimal value of binary number encoding the state of 16 inputs.

ReadDigitalOutputWord()

```
extern "C" WINAPI BOOL ReadDigitalOutputWord(
                               int iBoardIndex,
                               int * pOutputWord);
```

The `ReadDigitalOutputWord` function reads the current states of all 16 digital outputs of one digital I/O board. It takes two arguments, one which specifies the digital I/O board to be read and one to return an integer value encoding the states of the outputs. The function returns `TRUE` if the output word could be read, or `FALSE` if an error occurred.

Parameters

`iBoardIndex`	Index of digital I/O board.
`pOutputWord`	Pointer to integer value, used to return decimal value of binary number encoding the state of 16 outputs.

SetDigitalOutputWord()

```
extern "C" WINAPI BOOL SetDigitalOutputWord(
                                int iBoardIndex,
                                int iOutputWord);
```

The `SetDigitalOutputWord` function sets all 16 digital outputs of a digital I/O board at once. It takes two arguments, the first one specifying the digital I/O board, the second one holding the values of the bits. The function returns `TRUE` if the outputs could be set, or `FALSE` if an error occurred.

Parameters

`iBoardIndex`	Index of digital I/O board.
`iOutputWord`	State values to be set.

ReadFieldBusInputBit()

```
extern "C" WINAPI BOOL ReadFieldBusInputBit(
                                int iBoardIndex,
                                int iInputNumber);
```

The `ReadFieldBusInputBit` function reads the current state of an input bit of a field bus device. It takes two arguments which specify the field bus board and the input bit to be read. The function returns the state of the input bit, i.e. `TRUE` or `FALSE`.

Parameters

`iBoardIndex`	Index of field bus board.
`iInputNumber`	Number of input bit to be read.

SetFieldBusOutputBit()

```
extern "C" WINAPI BOOL SetFieldBusOutputBit(
                                int iBoardIndex,
                                int iOutputNumber,
                                BOOL bNewState);
```

The `SetFieldBusOutputBit` function sets an output bit of a field bus device. It takes three arguments, specifying the field bus board, the output bit to be set and the state the output bit will be set to. The function returns `TRUE` if the output bit was set successfully, or `FALSE` if an error occurred.

Parameters

`iBoardIndex`	Index of field bus board.
`iOutputNumber`	Number of output bit to be set.
`bNewState`	New state value.

ReadFieldBusInputImage()

```
extern "C" WINAPI BOOL ReadFieldBusInputImage(
                                    int iBoardIndex,
                                    int iByteCount,
                                    BYTE * pbyInputImage);
```

The `ReadFieldBusInputImage` function reads the current input image (input state) of a field bus device. It takes three arguments, the first one specifiying the field bus board, the second one the number of bytes (1 byte = 8 bits) to be read, the third one a pointer to a byte array to be filled with the current input bytes. The function returns `TRUE` if the input image could be read, or `FALSE` if an error occurred.

Parameters

`iBoardIndex`	Index of field bus board.
`iByteCount`	Number of bytes to be read = length of input image buffer.
`pbyInputImage`	Pointer to input image data (length of buffer = `iByteCount`).

ReadFieldBusOutputImage()

```
extern "C" WINAPI BOOL ReadFieldBusOutputImage(
                                    int iBoardIndex,
                                    int iByteCount,
                                    BYTE * pbyOutputImage);
```

The `ReadFieldBusOutputImage` function reads the current output image (output state) of a field bus device. It takes three arguments, the first one specifiying the field bus board, the second one the number of bytes (1 byte = 8 bits) to be read, the third one a pointer to a byte array to be filled with the current output bytes. The function returns `TRUE` if the output image could be read, or `FALSE` if an error occurred.

Parameters

`iBoardIndex`	Index of field bus board.
`iByteCount`	Number of bytes to be read = length of output image buffer.
`pbyOutputImage`	Pointer to output image data (length of buffer = `iByteCount`).

SetFieldBusOutputImage()

```
extern "C" WINAPI BOOL SetFieldBusOutputImage(
                              int iBoardIndex,
                              int iByteCount,
                              BYTE * pbyOutputImage);
```

The `SetFieldBusOutputImage` function sets a new output image (output state) of a field bus device. It takes three arguments, the first one specifiying the field bus board, the second one the number of bytes (1 byte = 8 bits) to be set, the third one a pointer to a byte array filled with the new values of the output bytes. The function returns `TRUE` if the output image could be set, or `FALSE` if an error occurred.

Parameters

`iBoardIndex`	Index of field bus board.
`iByteCount`	Number of bytes to be set = length of output image buffer.
`pbyOutputImage`	Pointer to new output image data (length of buffer = `iByteCount`).

4.2 Bitmap Management

Bitmap formats are essential for displaying images in Windows® applications. In order to free the plug-in DLL programmer from implementing these formats himself, the API provides several functions which allow convenient management of bitmaps. A bitmap is passed as bitmap handle which basically can be seen as pointer to a bitmap object. The bitmap objects created by the NeuroCheck API are kept within the API DLL. The API also provides functions to retrieve certain information about the existing bitmap objects. In order to release bitmap objects not used any longer, the respective API functions must be used. In any case all objects are released upon unloading `Ncapi.dll`.
For convenient management of bitmaps the API provides the following functions.

CreateMonoBitmap()

```
extern "C" WINAPI HBITMAP CreateMonoBitmap(
                      int iWidth,
                      int iHeight,
                      int iZoom,
                      BYTE * pbyGrayValues,
                      BOOL bOverlayColors);
```

The `CreateMonoBitmap` function can be used to create a 256 color bitmap. It takes five arguments, two specifying the dimensions of the image, one specifying the scale factor, one containing the pixel values of the gray level image, and one to enable the drawing of overlays. The function returns a handle to the created bitmap, or `NULL` on error.

Parameters

`iWidth`	Width of image.

`iHeight`	Height of image.
`iZoom`	Scale factor. Can have one of the following values, declared as symbolic constants in `pi_types.h`: `NC_ZOOM_10` 10 % of original image size. `NC_ZOOM_25` 25 % of original image size. `NC_ZOOM_50` 50 % of original image size. `NC_ZOOM_100` 100 % of original image size. `NC_ZOOM_200` 200 % of original image size.
`pbyGrayValues`	Pointer to image data.
`bOverlayColors`	If `TRUE`, values in the color palette are reserved for overlay colors. This is necessary for drawing on the image using the graphics device interface (GDI) of Windows®. Please refer to sample DLL for an example.

CreateColorBitmap()

```
extern "C" WINAPI HBITMAP CreateColorBitmap(
                                int iWidth,
                                int iHeight,
                                int iZoom,
                                BYTE * pbyRedData,
                                BYTE * pbyGreenData,
                                BYTE * pbyBlueData);
```

The `CreateColorBitmap` function can be used to create a TrueColor bitmap. It takes six arguments, two specifying the dimensions of the image, one specifiying the scale factor, three containing the pixel values of each color channel. The function returns a handle to the created bitmap, or `NULL` on error.

Parameters

`iWidth`	Width of image.
`iHeight`	Height of image.
`iZoom`	Scale factor. Can have one of the following values, declared as symbolic constants in `pi_types.h`: `NC_ZOOM_10` 10 % of original image size. `NC_ZOOM_25` 25 % of original image size. `NC_ZOOM_50` 50 % of original image size. `NC_ZOOM_100` 100 % of original image size. `NC_ZOOM_200` 200 % of original image size.
`pbyRedData`	Pointer to image data (red channel).
`pbyGreenData`	Pointer to image data (green channel).
`pbyBlueData`	Pointer to image data (blue channel).

DeleteBitmap()

```
extern "C" WINAPI BOOL DeleteBitmap(HBITMAP hBmp);
```

The `DeleteBitmap` function deletes the bitmap of the given handle. The function returns `TRUE` on success, `FALSE` on error.

Parameters

`hBmp`	Handle of bitmap to be deleted.

DeleteAllBitmaps()

```
extern "C" WINAPI void DeleteAllBitmaps(void);
```

The `DeleteAllBitmaps` function deletes all bitmap handles currently kept in the `Ncapi.dll`. The function has no parameters and no return value.

GetBitmapWidth()

```
extern "C" WINAPI int GetBitmapWidth(HBITMAP hBmp);
```

The `GetBitmapWidth` function reads the width of the given bitmap.
The function returns -1 on error.

Parameters

`hBmp`	Handle of bitmap.

GetBitmapHeight()

```
extern "C" WINAPI int GetBitmapHeight(HBITMAP hBmp);
```

The `GetBitmapHeight` function reads the height of the given image.
The function returns -1 on error.

Parameters

`hBmp`	Handle of bitmap.

GetBitmapNumColors()

```
extern "C" WINAPI int GetBitmapNumColors(HBITMAP hBmp);
```

The `GetBitmapNumColors` function returns the number of colors in color palette of bitmap. The function returns -1 on error.

Parameters

`hBmp`	Handle of bitmap.

Return Values

`0`	TrueColor bitmap
`256`	256 color bitmap

`-1`	Error.

LoadBitmapFromFile()

```
extern "C" WINAPI HBITMAP LoadBitmapFromFile(LPCTSTR lpszFilename);
```

The `LoadBitmapFromFile` function loads a specified bitmap file. The function returns the handle of the loaded bitmap on success, or `NULL` on error.

Parameters

`lpszFilename`	Name of bitmap file to be loaded.

SaveBitmapToFile()

```
extern "C" WINAPI BOOL SaveBitmapToFile(
                              HBITMAP hBmp,
                              LPCTSTR lpszFilename);
```

The `SaveBitmapToFile` function saves the given bitmap to file. The function returns `TRUE` on success, `FALSE` on error.

Parameters

`hBmp`	Handle of bitmap to be saved.
`lpszFilename`	Name of bitmap file.

GetAdvMemPtr()

```
extern "C" WINAPI BYTE* GetAdvMemPtr(unsigned int uiSize);
```

The `GetAdvMemPtr` function serves for passing large images faster to NeuroCheck. It reserves memory space of size `uiSize` for the image pointers in the `sPI_IMAGE` structure, i.e. for the struct members `pbyGrayValue`, `pbyRedValue`, `pbyGreenValue` and `pbyBlueValue`. The function returns the pointer address for the reserved memory block. This address is read-only and must be assigned to one of the aforementioned members of structure `sPI_IMAGE`. NeuroCheck then will use the pointer directly instead of copying the image data when using `VirtualAlloc` for allocation. Note that the structure of type `sPI_IMAGE` itself still must be allocated using `VirtualAlloc`. The function returns `NULL` on error.

Parameters

`uiSize`	Size of memory block to be reserved.

ReleaseAdvMemPtr()

```
extern "C" WINAPI BOOL ReleaseAdvMemPtr(BYTE* pbyData);
```

The `ReleaseAdvMemPtr` function releases memory space allocated with the `GetAdvMemPtr` function. The function returns `TRUE` on success, `FALSE` on error.

Parameters

`pbyData`	Address of memory block previously reserved with `GetAdvMemPtr`.

4.3 Access to NeuroCheck Settings

The API enables the plug-in DLL to access certain settings of NeuroCheck. For this purpose the API provides the following functions:

GetNcApiVersion()

```
extern "C" WINAPI int GetNcApiVersion(void);
```

The `GetNcApiVersion` function returns the version number of the NeuroCheck API as integer, e.g. 515. It will be increased whenever functionality is added to the DLL.

GetNcExeVersion()

```
extern "C" WINAPI BOOL GetNcExeVersion(
                           DWORD * pdwMajorVersion,
                           DWORD * pdwMinorVersion,
                           DWORD * pdwBuildVersion,
                           DWORD * pdwSubBuildVersion);
```

The `GetNcExeVersion` function reads the version numbers of the NeuroCheck main application. The version information, e.g. 5.1.1038.0, will be splitted into single parts that are returned by the argument pointers. The function returns `TRUE` on success, `FALSE` on error.

Parameters

`pdwMajorVersion`	Will be filled with major verson number, e.g. 5.
`pdwMinorVersion`	Will be filled with minor verson number, e.g. 1.
`pdwBuildVersion`	Will be filled with main build number, e.g. 1038.
`pdwSubBuildVersion`	Will be filled with sub build number, e.g. 0.

GetSecurityKeyItem()

```
extern "C" WINAPI int GetSecurityKeyItem(int iItemIndex);
```

The `GetSecurityKeyItem` function returns information about the security key of NeuroCheck. On error, the function returns `-1`.

Parameters

`iItemIndex`	Index of security key item. Can have one of the following values, declared as symbolic constants in `pi_NcApi.h`: `NC_KEY_LICENSE_NUMBER` return license number. `NC_KEY_LICENSE_LEVEL` return license level

As license level the function returns the same values as for the **LicenseLevel** property in the OLE automation interface (see 7.3.1 Properties of NCApplication Object).

GetCrCommId()

```
extern "C" WINAPI int GetCrCommId(void);
```

The `GetCrCommId` function returns the value of the check routine identification number (CRID) of the currently loaded check routine. If no check routine is loaded or upon error the function returns `-1`.

GetAppIoIgnoreFlag()

```
extern "C" WINAPI BOOL GetAppIoIgnoreFlag(void);
```

The `GetAppIoIgnoreFlag` function returns the value of the ignore cummunication option of NeuroCheck. If it returns `TRUE`, NeuroCheck currently ignores all functions requiring signals to be received via digital I/O or field bus; The default is `FALSE`.

SetCheckRoutineModifiedFlag()

```
extern "C" WINAPI void SetCheckRoutineModifiedFlag(BOOL bState);
```

The `SetCheckRoutineModifiedFlag` function modifies the "Dirty" flag of the current check routine. If this flag is `TRUE`, then upon closing the check routine NeuroCheck will display a message box asking if the user wants to save the check routine or not. The flag generally is set to `TRUE` in NeuroCheck if the check routine has been altered.

Parameters

`bState`	New state of "Dirty" flag.

GetParentOID()

```
extern "C" WINAPI int GetParentOID(int iOID);
```

The `GetParentOID` function returns the object identification number (OID) of the parent object for the object specified by the OID number. For a check function, the parent object is the single check it belongs to, for a single check the check routine. If there is no object with the specified OID, or if this object is the check routine and thus has no parent, then the function returns -1.

Parameters

`iOID`	Identification number of NeuroCheck object to be investigated.

GetXMLStringLength()

```
extern "C" WINAPI int GetXMLStringLength(int iOID, int iExportOptions);
```

The `GetXMLStringLength` function returns the length of the XML export string for the NeuroCheck object specified by its object identification number (OID), including the zero termination character. So the return value can be used immediately to allocate the string buffer for `GetXMLString`. If there is no object with the specified OID, or if the XML export fails, then the function returns -1.

Parameters

`iOID`	Identification number of NeuroCheck object to be investigated.
`iExportOptions`	Export options. See `GetXMLString`.

GetXMLString()

```
extern "C" WINAPI BOOL GetXMLString(
                                int iOID,
                                char* pszXMLString,
                                unsigned int uiMaxStringLength,
                                int iExportOptions);
```

The `GetXMLString` function returns the XML export string for the NeuroCheck object specified by its object identification number (OID). The string will be copied to the string buffer which must be allocated prior to calling this function. The exact buffer size can be determined with `GetXMLStringLength`. If the buffer is not large enough, the XML string will simply be truncated. The function returns `TRUE` on success, `FALSE` on error.

Parameters

`iOID`	Identification number of NeuroCheck object to be investigated.
`pszXMLString`	Pointer to previously allocated string buffer.
`uiMaxStringLength`	Maximum length of string that can be copied to string buffer.
`iExportOptions`	Export options. Can be any combination of the following values, declared as symbolic constants in `pi_NcApi.h`:

NC_XML_EXPORT_	
CHILDREN	main XML child elements
PROPERTIES	all other XML child elements
BINARY	binary data elements
DESCRIPTION	description elements (HTML)
ATTRIBUTES	supplementary attributes

The export options can be used to optimize execution speed. If only the parameters of a check function are needed including binary data (e.g. parameter block of plug-in functions), then `iExportOptions` could be set to
`NC_XML_EXPORT_CHILDREN + NC_XML_EXPORT_BINARY`

LogMessage()

```
extern "C" WINAPI BOOL LogMessage(
                               int iWarningLevel,
                               LPCTSTR lpszSource,
                               LPCTSTR lpszMsg);
```

The `LogMessage` function writes a message to the NeuroCheck log file. It can be used to easily output status or warning messages within the context of standard NeuroCheck messages and thus is very convenient for debugging a plug-in DLL. The message will only be written if logging is activated in NeuroCheck. The function returns `TRUE` on success, `FALSE` on error or if no logging is activated.

Parameters

`iWarningLevel`	Warning level, 1-4 from fatal error to simple information. Symbolic constants for the levels are declared in `PI_NcApi.h`.
`lpszSource`	Name of source, e.g. name of plug-in DLL of function.
`lpszMsg`	Message to be written.

4.4 Error Codes

The API provides following function to retrieve the most recent error that occured for a function call to NeuroCheck API.

GetLastNcApiError()

```
extern "C" WINAPI int GetLastNcApiError(void);
```

The `GetLastNcApiError` function returns a value indicating the most recent error that occured for a function call to NeuroCheck API. A call to `GetLastNcApiError` resets the internal error state to -1.

Return Value	Meaning
`-1`	No error.
`100`	No access to NeuroCheck application window.

`101`	Invalid parameter.
`102`	Invalid buffer size.
`110`	Error for API function `GetDeviceCount`.
`111`	Error for API function `GetSubDeviceCount`.
`120`	Error reading digital input.
`121`	Error setting a digital output.
`130`	Error setting output word for digital IO board.
`131`	Error reading input word for digital IO board.
`132`	Error reading output word for digital IO board.
`140`	Error reading digital input bit for field bus board.
`141`	Error setting a digital output bit for field bus board.
`150`	Error setting output image for field bus board.
`151`	Error reading input image for field bus board.
`152`	Error reading output image for field bus board.
`160`	Error setting modified flag for check routine.
`161`	Error reading ignore communication option of application.
`162`	Error allocating advanced memory pointer.
`163`	Error releasing advanced memory pointer.
`165`	Error or wrong argument in `GetSecurityKeyItem`.
`170`	Error creating bitmap.
`171`	Invalid bitmap handle.
`172`	Error loading bitmap from file.
`173`	Error creating bitmap file.
`174`	Error writing bitmap to file.
`175`	Invalid bitmap format.
`180`	Could not get version info of NeuroCheck executable.
`181`	Could not get object ID.
`182`	Error writing log file entry.
`183`	Error for API function `GetDeviceName`.

`184`	Error for API function `GetSubDeviceName`.
`185`	Error getting XML string.
`186`	Error getting XML string length.
`187`	Object identifier not found.
`188`	Check routine object has no parent.
`189`	Logging disabled.

5 Implementing a Plug-In DLL

The implementation of a plug-in DLL is demonstrated by several sample plug-in DLLs. The sample DLLs have been programmed using Microsoft® Visual C++ 6.0, but you can of course use any language and / or compiler which supports DLL compilation for implementing a plug-in DLL. Each project is contained in an own folder.

5.1 Overview of the Sample Plug-In DLLs

The samples can be installed or copied from the NeuroCheck CD-ROM. If installed from CD-ROM, the sample projects can be found in the folder `\Programming\PlugIn` in the NeuroCheck installation path.

PI_Simple

This is a very basic sample. The sample DLL only contains one plug-in check function. The sample also shows how to display an info dialog for a plug-in DLL.

- **Modify Image**
  The plug-in check function simply takes an input image and modifies its gray level pixel values. It has no parameters, no custom visualization and no data output capabilities.

PI_Menu

This sample demonstrates the use and implementation of plug-in menu commands. It implements three plug-in menu commands to be called in manual mode and three others to be called in automatic mode. The plug-in menu commands are appended to the **Tools** menu in NeuroCheck for the respective mode. The sample DLL does not register any plug-in check function.

PI_Parameter

This sample shows the usage of a parameter set and the implementation of a parameter dialog for a plug-in check function. It also shows the usage of help files for plug-in check functions. For this and all following samples, each plug-in check function is encapsulated in its own source file.

- **Create New Image**
  This plug-in check function demonstrates the use of parameters in a plug-in check function and the implementation of a simple parameter dialog. It also demonstrates the creation of a new data object.
  The plug-in check function takes an input image and creates an output image which is the mirror of the input image. If the parameter flag is `TRUE`, the output image additionally will be inverted.

- **Modify Histogram**
  This plug-in check function demonstrates the implementation of a parameter dialog which

accesses input data of the function. An Update button can be used to dynamically update the input data for a new image.
The plug-in check function takes a histogram as input object and modifies the threshold value.

- **Modify Color Image**
  This plug-in check function demonstrates the use of a structure as parameter block. It gives detailed hints for the setting of default values and version management in the initialization routine.
  The plug-in check function takes a color image as input object and inverts one of its color channel according to the option chosen in its parameter dialog. It aborts execution if the input image has no color information. In manual mode, in this case a message box is displayed to inform the user.

PI_DataTypes

This sample shows how to access, modify and create output objects of different NeuroCheck data types. Please refer also to the functions of `PI_Parameter` (e.g. Modify Histogram). Examples for more complicated operations and for the usage of plug-in data types are given in `PI_DataTypes_Adv`.

- **Create Histogram**
  This plug-in check function demonstrates the creation of a `PI_HISTO` data type object. It also shows the access of image and layer data. Another example for creating a histogram can be found in `PI_DataOutput`.

- **Modify Layer**
  This plug-in check function demonstrates the modification of an object layer. In detail following modifications are shown:
- ordering of objects (modification of object number)
- grouping of objects (modification of group number)
- activating and calculating feature values (`fMeasurement` array)
- altering position of objects
- declaration of invalid objects (`bValid` flag)

- **Create MeasArray**
  This plug-in check function demonstrates the creation of a `PI_MEASARRAY` data type object. It also shows the read access of the feature values of a layer object. For each activated feature, the sum is calculated for all objects in the input layer.
  Another example for creating a meas array can be found in `PI_DataOutput`.

- **Modify MeasArray**
  This plug-in check function demonstrates the modification of a measurement array. Each measurement value is replaced by its square value.

PI_DataTypes_Adv

This sample demonstrates the use of plug-in data types. It also gives advanced examples for the use of NeuroCheck data types. Less complex examples are given in `PI_DataTypes` and `PI_DataOutput`.

- **Create Plug-In Data Types**
  This plug-in check function demonstrates the creation of plug-in data types. Furthermore it shows the access of class information of objects and of source information of images.

- **Modify Plug-In Data Type**
  This plug-in check function demonstrates the modification of a plug-in data type. The plug-in check function takes as input an object of plug-in data type `DT_03`. It simply changes the CString object encapsulated in the plug-in data type.

- **Create Color Image**
  This plug-in check function demonstrates the creation of a color image. It also shows the read access of ROI descripitions, i.e. chain code for the contour and RLC code for the region description.
  A simpler example for creating a `PI_IMAGE` object can be found in `PI_DataOutput`.

- **Create Layer**
  This plug-in check function demonstrates the creation of layer objects and the access of model geometries. Objects in the input layer for which no model geometry has been calculated for simply will be copied to equivalent objects in the output layer if possible. For objects a model geometry has been calculated for an output object will be created matching the model geometry.
  A simpler example for creating a `PI_LAYER` object can be found in `PI_DataOutput`.

PI_DataOutput

This sample DLL shows how plug-in check functions can participate in NeuroCheck's file output capabilities

- **Output of Color Image**
  This plug-in check function demonstrates output of an image. It also gives a simple example for the creation of a color image.

- **Output of Plug-In Data Types**
  This plug-in check function demonstrates the creation and data output of plug-in data types.

- **Output of MeasArray**
  This plug-in check function demonstrates the creation and data output of a `PI_MEASARRAY` data type object.

- **Output of Layer**
  This plug-in check function demonstrates the creation and data output of a layer object. It creates one rectengular AOI and activates some feature values.

- **Output of Histogram**
  This plug-in check function demonstrates the creation and data output of a `PI_HISTO` data type object.

PI_NcApiCalls

This sample shows the usage of the NeuroCheck API functionality provided in `NcApi.DLL`. A further example is shown in `PI_Visualization`. For this sample, the NeuroCheck API is encapsulated in a C++ class available for re-use.
Please note that for successful execution of this sample, the check box „Provide API functionality" on page Plug-In of the General Software Settings dialog box must be activated.

- **Read Digital Input Word**
  This plug-in check function demonstrates the reading of an input word from a digital I/O board.

- **Set Digital Output Word**
  This plug-in check function demonstrates the setting of an output word for a digital I/O board.

- **Save Bitmap**
  This plug-in check function demonstrates the creation of a bitmap object and the saving of the bitmap to file using the NcApi functionality. For extended examples on creating bitmaps please refer to `PI_Visualize`.

- **Update Counter**
  This plug-in check function creates a meas array with a single measurement containing the value of the current count value. It demonstrates the setting of the modified flag of the current check routine.
  The function has a parameter set which is updated for each execution of the plug-in check function. In order to save the parameter block upon closing NeuroCheck or for a type change, the modified flag is set to `TRUE`.

- **Modify Field Bus Output Image**
  This plug-in check function demonstrates the access of a field bus device using the NcApi functionality. It shifts the bytes of the output image.

- **Create Fast Color Image**
  This plug-in check function demonstrates the creation of an image using the NcApi function GetAdvMemPtr().

PI_Visualization

This sample demonstrates the usage of custom result views. It also shows further examples for using the NeuroCheck API functionality. The NeuroCheck API is encapsulated in a C++ class available for re-use.
Please note that for successful execution of this sample, the check box „Provide API functionality“ on page Plug-In of the General Software Settings dialog box must be activated.

- **Visualize Images**
  The plug-in check function creates a color image and displays each color channel in a own result view. It also displays the input image as gray level image. The function also demonstrates the usage of the NeuroCheck API functionality to create bitmaps.

5.2 Project Structure

A plug-in DLL for NeuroCheck is created within a standard Visual C++ project for DLL programming. It will usually be unnecessary to make any changes to the core project files, excepting dialog and menu ressources. Therefore we recommend to reuse the files from one of the sample projects meeting your requirements. First create a new project of type `MFC AppWizard(dll)` using the Microsoft Foundation Classes as a static library. The AppWizard will automatically create several files, whose contents should then be replaced by that of the corresponding files from a sample project.
The file names are derived from the project name. For the example project called `PI_xxx` these are the following files:

File	Description
`PI_xxx.def`	Definition file, exports the required administrative functions. Do not change unless additional administration functions are required.
`PI_xxx.cpp`	Initialization routines, do not change.
`PI_xxx.h`	Header file, do not change.
`PI_xxx.rc`	Ressource file, contains the mandatory info dialog, which can be adapted to the desired look-and-feel or your plug-in DLL. Parameter dialogs of individual functions can be defined here as well.
`PI_Types.h`	Header file with data types and constants of the NeuroCheck plug-in interface. Do not change.
`PI_Main.cpp`	Administrative functions. This is the base implementation file to be adapted. We recommend to modularize your project such that each plug-in check function is encapsulated in a single module, i.e. header and implementation file. All samples with the exception of the `PI_Simple` project demonstrate this principle.

The next two files are only part of projects `PI_NcApiCalls` and `PI_Visualization`.

`PI_NcApi.h`	Header file with wrapper class definition, data types and constants for the NeuroCheck API interface. Do not change.
`PI_NcApi.cpp`	Implementation of wrapper class for conventient usage of the NeuroCheck API interface. Do not change.

NOTE: Do not use the MFC as a DLL, as this would conflict with NeuroCheck and cause the DLL loading process to fail. Instead it is recommended to statically link the MFC. This has the additional advantage that it is not necessary to check the version for the installed MFC DLL when using the plug-in DLL on a different system.

6 Custom Communication Interface

NeuroCheck allows you to seamlessly integrate your own communication interface DLL. It will work exactly as the serial interface DLL included with the setup.

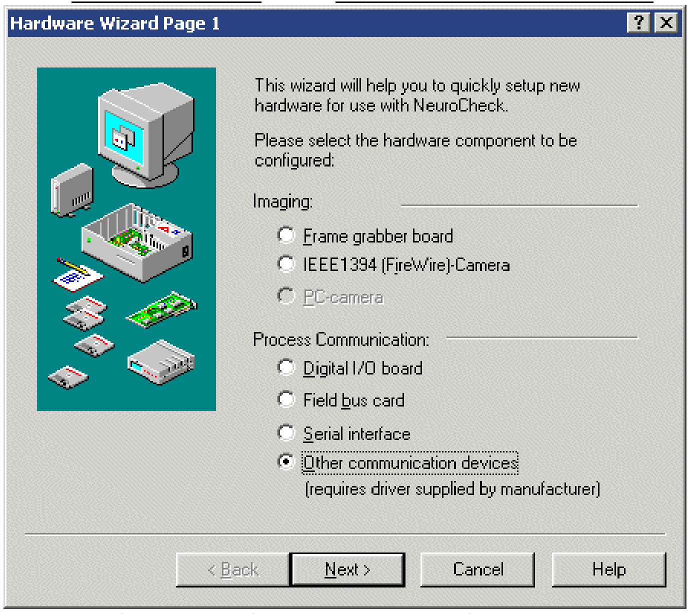
Implementation Considerations

The function declarations described in the following refer to Microsoft® 32 bit C/C++ compilers. When using a different compiler take care to use compatible data types and parameter passing methods. Furthermore it has to be ensured that the DLL exports all function names explicitly.
A Visual C++® sample project can be installed or copied from the NeuroCheck CD-ROM. If installed from CD-ROM, you will find it in the folder `\Programming\CustComm` in the NeuroCheck installation path. Please note that it is more a template to base your own implementation on than a ready-to-use example.

6.1 Using a Custom Communication Interface

6.1.1 Loading a Custom Communication DLL

Like all device drivers in NeuroCheck a custom communication DLL is loaded using the Hardware Wizard invoked by choosing New in the Device Manager dialog. On the first page of the Hardware Wizard select Other communication devices and choose Next.



Hardware Wizard Page 1

This wizard will help you to quickly setup new hardware for use with NeuroCheck.

Please select the hardware component to be configured:

Imaging:

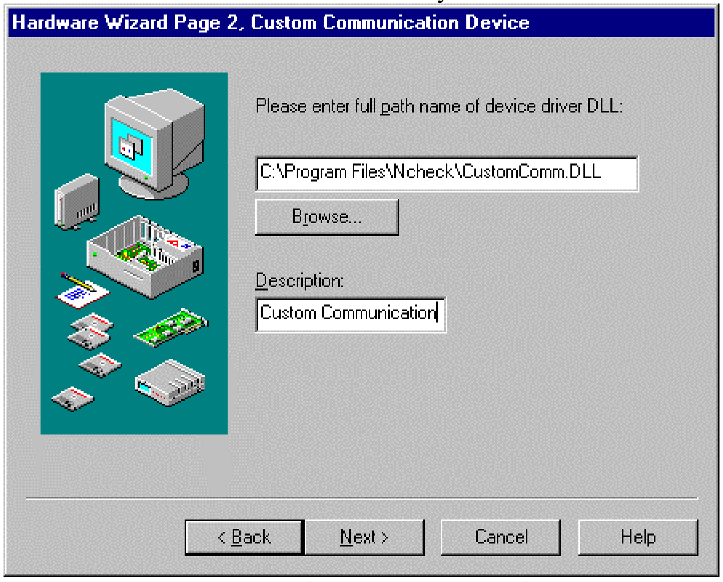
- ( ) Frame grabber board
- ( ) IEEE1394 (FireWire)-Camera
- ( ) PC-camera

Process Communication:

- ( ) Digital I/O board
- ( ) Field bus card
- ( ) Serial interface
- (●) Other communication devices (requires driver supplied by manufacturer)

Figure 9: first page of hardware wizard for custom communication

On the next page you have to enter the full path name of the device driver DLL (or select it via the Browse button). Then you have to enter a descriptive string. This string is used in the Device Manager to designate the interface and is mandatory. Note that only a single custom communication DLL can be loaded at any time.


Hardware Wizard Page 2, Custom Communication Device

Please enter full path name of device driver DLL:

C:\Program Files\Ncheck\CustomComm.DLL

Description:

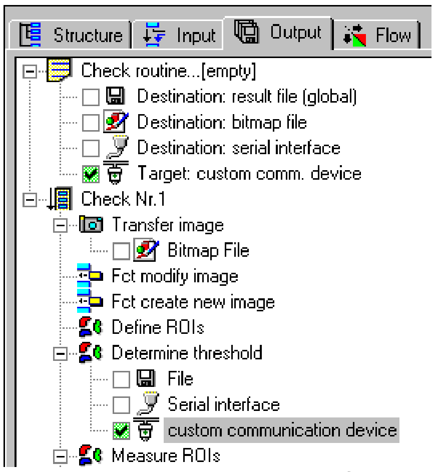
Custom Communication

Figure 10: second page of hardware wizard for custom communication

The final page of the Hardware Wizard merely informs you about the settings you have made on the previous pages. Choosing Finish loads and tests the device driver DLL.

6.1.2 Transmitting Data via the Custom Communication Interface

In order to transmit data via the custom communication interface it has to be activated globally by checking the Destination: custom comm. device box on the Output page of the check routine window.

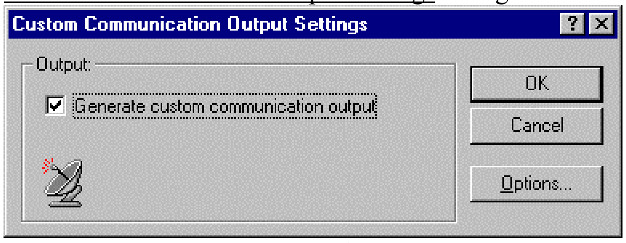


Check routine...[empty]
- ☐ Destination: result file (global)
- ☐ Destination: bitmap file
- ☐ Destination: serial interface
- ✓ Target: custom comm. device

Check Nr.1
- Transfer image
  - ☐ Bitmap File
- Fct modify image
- Fct create new image
- Define ROIs
- Determine threshold
  - ☐ File
  - ☐ Serial interface
  - ✓ custom communication device
- Measure ROIs

Figure 11: global activation of custom communication on output tab page

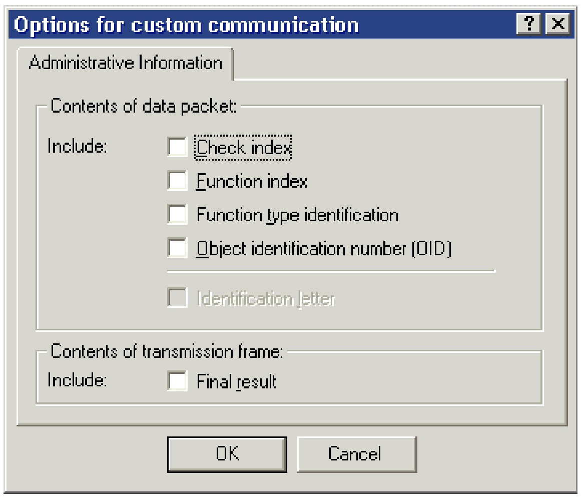
Alternatively you can choose **Data Output ► Custom Communication...** from the **Check Routine** menu and activate the Generate custom communication output check box in the Custom Communication Output Settings dialog box.



[✓] Generate custom communication output

Figure 12: global activation of custom communication via menu command

This dialog is also invoked by choosing **Change Settings** from the context menu of the Destination: custom comm. device box on the Output page of the check routine window. Choosing the Options button in this dialog invokes the Options for Custom Communication dialog, where you can activate several administrative information items to be included in the transmission. This dialog equals the one for serial communication in NeuroCheck. NeuroCheck treats the custom communication interface as a serial device, because this is the most universal output device, as it can be used for remote control as well as for data output.


Options for custom communication

Administrative Information

Contents of data packet:

Include:
- [ ] Check index
- [ ] Function index
- [ ] Function type identification
- [ ] Object identification number (OID)
- [ ] Identification letter

Contents of transmission frame:

Include:
- [ ] Final result

OK Cancel

Figure 13: custom communication options

For details on these options refer to the online help system of NeuroCheck.

6.1.3 Using the Custom Communication Interface for Remote-Control

Like the standard serial interface protocol the custom communication interface can be used for remote-control of NeuroCheck in automatic mode.

Start Check

In order to initiate a check routine run by a signal from the custom communication interface choose **Remote Control** from the **System** menu, switch to the Input page, activate the Start check signal, choose Change and select the Custom communication interface indicated by the antenna icon from the Select Signal Source dialog box. NeuroCheck will then poll the custom communication interface using function `TestStart()` while it waits for a start signal.

Select Check Routine

In order to switch check routines automatically by a signal from the custom communication interface choose **Remote Control** from the **System** menu, switch to the Input page, activate the Select check routine signal, choose Change and select the Custom communication interface indicated by the antenna icon from the Select Signal Source dialog box. NeuroCheck will then poll the custom communication interface using function `GetTypeId()` while it waits for a check routine selection signal.

Transmit Check Result

In order to have the final result of a check routine run automatically transmitted via the custom communication interface choose **Remote Control** from the **System** menu, switch to the Output page, activate the Check result signal, choose Change and select the Custom communication interface indicated by the antenna icon from the Select Signal Destination dialog box. NeuroCheck will then invoke function `SetCheckResult()` as soon as a check routine run has been completed.

6.2 Administrative Functions

6.2.1 Driver Initialization

Upon loading the driver (from the Hardware Wizard or during program start-up) NeuroCheck calls an initialization function. The programmer of the communication DLL can use this initialization function to check for the presence of the required communication hardware, open communication channels etc. The function is declared as:

```
extern "C" BOOL DllInit (void)
```

It has to return `1` for a successful initialization, `0` else. The function must be exported explicitly from the DLL.

6.2.2 Driver Setup

The DLL can contain and export a function declared as follows:

```
extern "C" BOOL SetupDevice (HWND hwndAppMain)
```

If this function exists and the user chooses "Properties" in the Device Manager while the custom communication interface is selected, NeuroCheck will call this function. It receives a handle to the main application window so that the programmer of the communication DLL can display his own dialog for setting up the device, analogous to the setup dialog of the serial interface DLL included with NeuroCheck.
The return value of the function indicates the return status of the setup dialog. If the function returns `TRUE`, NeuroCheck will call function `DllInit()` again.
If the function does not exist, NeuroCheck will not react to the "Properties" button for a custom communication interface.

6.2.3 Driver Test

The DLL can contain and export a function declared as follows:

```
extern "C" BOOL TestDevice (HWND hwndAppMain)
```

If this function exists and the user chooses "Test" in the Device Manager while the custom communication interface is selected, NeuroCheck will call this function. It receives a handle to the main application window so that the programmer of the communication DLL can display his own dialog for testing the device, analogous to the test dialog of the serial interface DLL included with NeuroCheck. If the function does not exist, NeuroCheck will not react to the "Test" button for a custom communication interface (the same holds for the **Test Custom Communication** item from the **System** menu)
The function should return `TRUE` if the device is working properly, `FALSE` else.

6.3 Remote Control Functions

6.3.1 Test for Start Signal

When NeuroCheck is in automatic mode and no check routine is running it polls the device selected for the Start check signal in the Remote Control dialog periodically. If this device is a custom communication interface it will call a function declared as follows:

```
extern "C" BOOL TestStart (void)
```

As long as this function returns `0` NeuroCheck continues polling. Upon a return value of `1` NeuroCheck will start the check routine (or wait for a check routine selection signal; see below). The function must be exported explicitly from the DLL.
NOTE: this function can be used to start a check a predefined number of times automatically by using an internal counter to decide, whether to return `1` or `0`.

6.3.2 Retrieve Check Routine Selection Signal

When NeuroCheck has received a Start check signal in automatic mode and the Check routine selection signal is activated in the Remote Control dialog it will poll the device selected for this signal periodically. If this device is a custom communication interface it will call a function declared as follows:

```
extern "C" BOOL GetTypeId (unsigned short int * pTypeId)
```

As long as this function returns `0` NeuroCheck continues polling. Upon a return value of `1` NeuroCheck will try to load the check routine with the identification number returned in `*pTypeID`. The function must be exported explicitly from the DLL.
The valid range for check routine identification numbers is `0` to `99999`, i.e. every positive integer that can be represented by five digits.

6.4 Result Output Functions

6.4.1 Final Check Result

After completing a check routine run NeuroCheck will send the final result of the check (OK or not OK) to the device selected for the Check result signal in the Remote Control dialog box. If this device is a custom communication interface (or if the option to include the check result in the transmission frame has been activated in the Options for Custom Communication dialog) it will call a function declared as:

```
extern "C" void SetCheckResult (BOOL bSuccess)
```

If the check result is OK, a value of `1` will be passed in `bSuccess`, a value of `0` else. The function must be explicitly exported from the DLL.

6.4.2 Floating Point Value

Functions like Measure ROIs or Gauge ROIs compute floating point values. If a custom communication interface has been selected as destination device for the results of such a function, NeuroCheck will call a function `TransferFloat()` for every value to be transmitted. The function is declared as:

```
extern "C" void TransferFloat (float fValue)
```

In `fValue` NeuroCheck passes the floating point value to be transmitted.

6.4.3 Integer Value

Functions like Count ROIs or Determine threshold compute integer values. If a custom communication interface has been selected as destination device for the results of such a function, NeuroCheck will call a function `TransferInt()` for every value to be transmitted. The function is declared as:

```
extern "C" void TransferInt (int iValue)
```

In `iValue` NeuroCheck passes the integer value to be transmitted.

6.4.4 String

Functions like Classify ROIs or Identify bar code compute string values. If a custom communication interface has been selected as destination device for the results of such a function, NeuroCheck will call a function `TransferString()` for every value to be transmitted. The function is declared as:

```
extern "C" void TransferString(char * pChar, unsigned int uiNumOfChars)
```

In `pChar` NeuroCheck passes the starting address of the string to be transmitted, in `uiNumOfChars` the number of valid characters.

6.4.5 Actuating Transmission

For performance reasons it may be advisable not to perform an actual transmission in the transfer functions described above but instead to buffer the data passed to these functions internally in the DLL. To enable programmers to optimize their DLL in such a way NeuroCheck calls a `Flush()` function at the end of a check routine run, when any output has been directed to a custom communication interface. The function is declared as:

```
extern "C" BOOL Flush (void)
```

The programmer can then use this function to perform the actual transmission. A return value of `1` indicates to NeuroCheck that communication has been executed without problems, whereas a return value of `0` indicates communication failure. If a system log output window exists on the automatic screen NeuroCheck will report this failure.

7 OLE Automation

This chapter documents the OLE automation interface exposed by NeuroCheck and contains excerpts from the NeuroCheck example automation controller to illustrate its use. For additional information on OLE technology please refer to Microsoft® operating system and development documentation, especially the documentation on "Array Manipulation Functions" regarding safearrays.

7.1 Introduction

7.1.1 What is OLE?

The original meaning of the acronym OLE is "**O**bject **L**inking and **E**mbedding" denoting a technology for exchanging data objects between applications. A typical example can be found in everyday office work: a chart from a spreadsheet program can be "embedded" in a word processor document so that a double-click on the chart will automatically activate the spreadsheet program for editing the chart. Alternatively, the chart can be linked into the document to allow automatic updating of the chart in the document whenever the original chart is updated in the spreadsheet program.

The original OLE functionality has later been extended into a general standard for the exchange of objects between applications. This standard refers to data objects as well as to programmable objects enabling programs to access the functionality of other programs. These programmable objects expose certain interfaces to the system allowing other OLE-capable applications to control the original programmable object. Controlling another application's programmable objects is called OLE automation. We will come to that in a moment.

First, though, a few remarks on COM and ActiveX, just to avoid confusion about all these names and acronyms. Details on the various technologies related to OLE can be found in Microsoft's OLE programmer's reference and a variety of other volumes.

COM stands for **C**omponent **O**bject **M**odel. This is the technology upon which OLE is built. COM specifies a set of rules for the creation of binary objects that can communicate with each other. By following these rules, programmable objects can be written in any OLE-capable programming language and accessed by any OLE-capable application. Recently, COM has been further extended to function between various computers connected over a network and is consequently now called DCOM, **D**istributed COM (DCOM). The most recent extension of COM/OLE bears the name ActiveX. Its most important aspect is the capability to exchange program functionality through the world-wide Internet. The distribution of programmable objects over the Internet allows for the integration of application capabilities into the hitherto rather static textual and graphical information on the World Wide Web.

7.1.2 What is OLE Automation?

Above we introduced the term *OLE automation* for the technology of controlling another application's objects. Like OLE, it is build on the COM techology and is therefore also called *ActiveX Automation*. In OLE automation there is a clear distinction between two different roles of programs:

- The application that supplies the programmable object is called the *OLE automation server*.
- The application using the functionality provided by the server is called the *OLE automation client*.

In effect, the *client* controls server. In this sense, the server is the slave, the client the master which may sound a bit confusing at first.

We have to distinguish further between two types of OLE automation servers:

- *In-process-servers* in the form of a dynamic link library (DLL); they are called *in-process* because the DLL is loaded into the address space of the controlling process.
- *Out-of-process-servers*, i.e. stand-alone executable files which expose an interface through which parts of their functionality can be controlled from the outside.

Since NeuroCheck obviously is a stand-alone executable file, it represents an out-of-process-server, which is why we will focus on this type of servers in the following.

A client must first connect to the server in order to use its functionality. This connection is established through the Windows® Automation Manager. This in turn queries the system registry for information about the server. Therefore, NeuroCheck must be started at least once as an independent executable in order to register itself as an OLE automation server before you can control it through a client.

☞ NeuroCheck will update the pertaining registry entry whenever it is started, so if you are running various versions of NeuroCheck on the same PC, the registry entry will always refer to the version most recently used.

After the connection has been established, the client can access the automation objects of the server. It may read or set properties or call methods of the objects. The complete communication is handled by the Automation Manager, i.e. each function call is handled through the operating system.

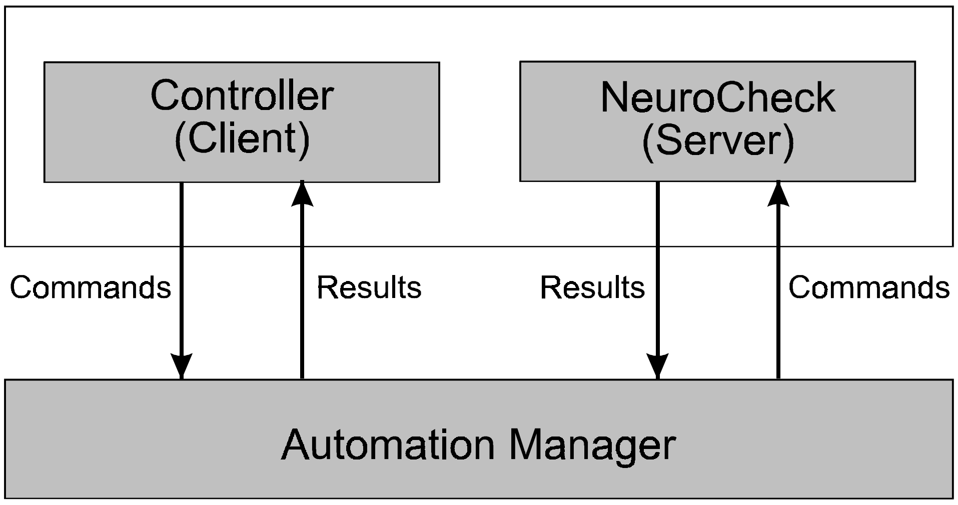


Figure 14: OLE Communication for out-of-process servers (stand-alone executables).

The exposed automation objects of NeuroCheck are listed in section 7.2 and described in detail in the subsequent sections. In the following we will first have a look at possible applications of using OLE automation with NeuroCheck.

7.1.3 Applications of OLE Automation in NeuroCheck

As an example, an OLE automation client connected to NeuroCheck can open a check routine, put NeuroCheck into automatic mode, execute the check routine, read the results of the inspection run and display them in a special way which is not available in NeuroCheck. Note that a controller is not restricted to connect to only a single server. It can thus control NeuroCheck and a database management system simultaneously and transfer result data computed by NeuroCheck into the database or control the operation of NeuroCheck based on information from the database.
The following list is intended to give you an idea of what can be achieved with NeuroCheck and OLE automation. This is of course not the end of it. The possibilities are practically without limit, as it is the express purpose of OLE automation to enhance each program's capabilities by borrowing functionality from other specialized programs. You can, for example:

- Design a specialized user interface for the production line personnel; this user interface might feature simplified input of a small subset of parameters relevant during production operation, give detailed instructions for certain events, or display custom statistics and charts;
- Use a program written in a standard programming language like Visual Basic® to perform communication and control tasks specific to your problem instead of applying a PLC;
- Hide NeuroCheck during production operation and still use it as a convenient development and test environment for the configuration of your inspection routines;
- Keep NeuroCheck on the screen to visualize the inspection process but prevent user interaction with NeuroCheck in automatic mode;
- Connect NeuroCheck to a database, e.g. for setting target values at runtime or storing error statistics;
- Integrate NeuroCheck into your process control system or communication setup;
- Implement a teach functionality specific to your inspection tasks;
- Use NeuroCheck as an intelligent sensor in a closed-loop control system realized by evaluating the results in your OLE controller.

7.1.4 Capabilities of the NeuroCheck OLE Automation Interface

This section will give you an idea of the extent of NeuroCheck's OLE automation interface. The exposed automation objects and their properties and methods are detailed in section 7.2 and following. You can control the following operations in NeuroCheck through OLE automation:

- load and save check routines,
- switch operating modes,
- execute a check routine and read its result,
- access image data,

- transfer complete images to other applications via clipboard,
- obtain information about the hierarchical structure of a check routine and alter designations like names, descriptions, comments, etc.
- activate or deactivate individual checks,
- alter parameters and target values of individual check functions,
- read result and status information for individual checks and check functions,
- read result values of individual check functions,
- obtain information about the currently running server, like version number, license level, license number, etc.
- change size, position, state and visibility of NeuroCheck's main window,
- get information about devices configured in NeuroCheck's device manager,
- read input and output states and set output bits of all digital I/O boards and field bus devices configured in NeuroCheck's device manager,
- select layouts for the automatic mode screen,
- select camera and zoom for live image display,
- get detailed diagnostic information.

7.1.5 Restrictions

In contrast to plug-in functions or custom communication which are integrated in NeuroCheck an OLE controller is an independent application. OLE automation can be seen as remote control of NeuroCheck. When controlled through OLE automation, NeuroCheck acts as a slave. The OLE client takes the role of the master which means that:

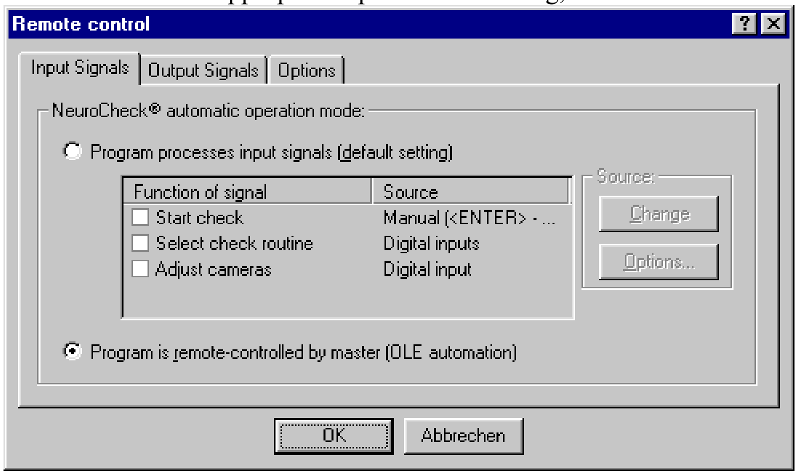
- The client takes the place of the usual user interface;
- The client is the sole entity allowed to initiate actions in NeuroCheck, like the execution of an inspection run or a change of check routines.

In effect, the client is responsible for the entire control and timing of the application. Therefore, the following restrictions apply when running NeuroCheck through OLE automation:

- Interactive use of NeuroCheck's application window is disabled.
- Input signals configured under **System/Remote Control** (start check, select check routine, adjust cameras) are ignored.
- Self test is not available.
- Operating modes "Test" and "Configure Automatic Screen" are not accessible (of course the usual restrictions related to a particular license level still apply; even using OLE automation you will not be able to switch a runtime version to "Manual" mode);
- Settings for start-up behavior made under **System/Options** are ignored.

7.1.6 Preparing NeuroCheck for Control by OLE Automation

As stated above, certain restrictions apply when NeuroCheck is controlled by an OLE client. This means that NeuroCheck must be configured explicitly to accept commands from an OLE client. This configuration takes place in the Remote Control dialog opened by the **Remote-Control** command from the **System** menu. For NeuroCheck to be controlled via OLE you have to activate the appropriate option in this dialog, as shown in the following figure.



Remote control

Input Signals | Output Signals | Options

NeuroCheck® automatic operation mode:

( ) Program processes input signals (default setting)

Function of signal	Source
☐ Start check	Manual (<ENTER> - ...
☐ Select check routine	Digital inputs
☐ Adjust cameras	Digital input

(●) Program is remote-controlled by master (OLE automation)

Figure 15: NeuroCheck must be configured to be controlled via OLE automation using the **Remote Control** command in the **System** menu

7.1.7 Programming Language

The COM specification is indepent of the programming language. This means that an OLE client can be written in any OLE-capable programming language, e.g. the various C/C++ environments, Visual Basic® or Borland Delphi®. Unless your controller needs to execute very complex algorithms, the decisive factor for the speed of execution will be the number of calls to the OLE interface due to the handling of OLE commands through the Automation Manager of the operating system.
Also important for the speed of your OLE application is the data exchange method used. You will see in section 7.7.1 that there are some interface functions using the SafeArray concept for data exchange. These functions can be used most effectively in C/C++. Although they can be applied in other languages, even pointer-challenged languages like Visual Basic®, other means of data exchange are much simpler to program, though less efficient.

7.1.8 Using Type Libraries

Type libraries contain information about data types, interfaces, member functions, and object classes exposed by an OLE server. NeuroCheck provides a type library called `NCheck.tlb`. It is essential for programming the OLE interface in C/C++, but can also be very helpful for Visual Basic® or Delphi® programmers. Including the type library in their projects enables early binding and automatic syntax checking by the compiler and allows for convenient use of NeuroCheck's on-line help within the development environment of the programming language.

7.2 Exposed Automation Objects

The following table lists the automation objects exposed by NeuroCheck. Subsequent sections explain the properties of these automation objects in detail.

Object name	Description
NCApplication	Top-level object; provides a standard way for OLE automation controllers to retrieve and navigate low-level objects.
CheckRoutine	Provides a way to change the settings of a NeuroCheck check routine.
SingleCheck	Gives access to the properties of individual checks.
CheckFunction	Gives access to the properties of check functions.

The following figure depicts this object hierarchy.

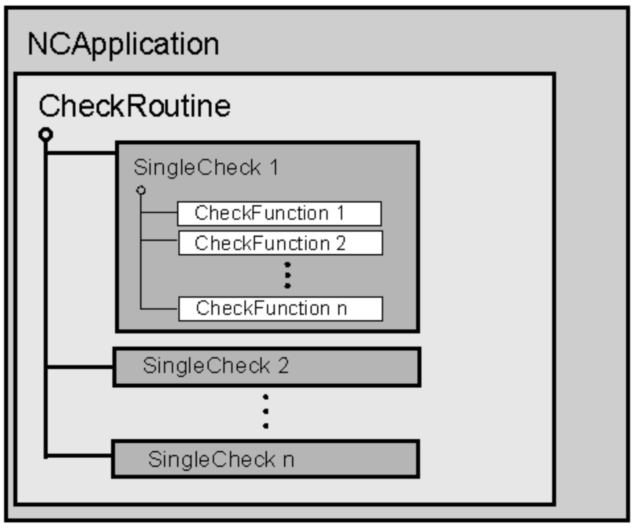


Figure 16: Hierachy of exposed automation objects

7.3 NCApplication Object

7.3.1 Properties

This section lists the properties of the **NCApplication** object. All constants are listed in `ncauto.h`.

Note: C/C++ programmers must access the properties as follows:

ptrObj→**Get***PropertyName***()** for reading a property

ptrObj→**Set***PropertyName***(***NewValue***)** for setting a property (not available for read only properties)

The *ptrObj* placeholder represents a pointer to the **NCApplication** object, *PropertyName* the name of the property and *NewValue* the new value the property is set to.

ActiveCamera

The **ActiveCamera** property sets or returns the index of the camera from which the image is captured in live mode.

Syntax

*object.***ActiveCamera** *[=value]*

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_I2

Remarks

The **ActiveCamera** property can only be accessed in live mode.

ActiveCameraName

The **ActiveCameraName** property returns the designation of the camera from which the live image is captured. It is only available in live mode. Camera designations correspond to those in the Device Manager dialog.

Syntax

*object.***ActiveCameraName**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_BSTR

ActiveCameraZoom

The **ActiveCameraZoom** property sets or returns the zoom factor of the live image view. It is only available in live mode.

Syntax

*object.***ActiveCameraZoom** *[=value]*

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_I2

Possible settings for **ActiveCameraZoom** are:

Constant	Value	Description
`NC_ZOOM_100`	`100`	100 % of original image size
`NC_ZOOM_50`	`50`	50 % of original image size
`NC_ZOOM_25`	`25`	25 % of original image size
`NC_ZOOM_10`	`10`	10 % of original image size

ActiveCheckRoutine

The **ActiveCheckRoutine** property returns the active check routine object or VT_EMPTY if none is available.

Syntax

*object.***ActiveCheckRoutine**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_DISPATCH

Application

The **Application** property returns the application object.

Syntax

*object.***Application**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_DISPATCH

Caption

The **Caption** property returns the title of the application window.

Syntax

*object.***Caption**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_BSTR

DeviceCount

The **DeviceCount** property returns the number of devices of the type specified by *DeviceType* configured in NeuroCheck's Device Manager. If the input argument is incorrect, the property returns -1.

Syntax

*object.***DeviceCount(***DeviceType***)**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_I2

Argument	Type	Description
DeviceType	VT_I2	Type of the device.

Possible settings for *DeviceType* are:

Constant	Value	Description
NC_DEVICE_FRAMEGRABBER	0	Frame grabber board
NC_DEVICE_DIGITALIO	1	Digital I/O board
NC_DEVICE_FIELDBUS	2	Field Bus board
NC_DEVICE_SERIAL	3	Serial Interface
NC_DEVICE_OEMCOM	4	Custom Communication Interface
NC_DEVICE_IEEE1394	5	IEEE 1394 camera

DeviceName

The **DeviceName** property returns the name of a device configured in NeuroCheck's Device Manager. The device is specified by the input arguments *DeviceType* and *DeviceIndex.* If the input arguments are incorrect, the property returns an empty string.

Syntax

*object.***DeviceName(***DeviceType, DeviceIndex***)**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_BSTR

Argument	Type	Description
DeviceType	VT_I2	Type of the device.
DeviceIndex	VT_I2	Index of the device (counted from 0).

The possible settings for argument *DeviceType* correspond to those for the **DeviceCount** property of the **NCApplication** object.

ExeMajorVersion

The **ExeMajorVersion** property returns the major version number of the application's executable file (always greater or equal to 4).

Syntax

*object.***ExeMajorVersion**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_I2

ExeMinorVersion

The **ExeMinorVersion** property returns the minor version number of the application's executable file (always greater or equal to 0).

Syntax

*object.***ExeMinorVersion**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_I2

FullName

The **FullName** property returns the file specification for the application, including path.

Syntax

*object.***FullName**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_BSTR

Height

The **Height** property returns or sets the distance between the top and bottom edge of the main application window. See Figure 17.

Syntax

*object.***Height** *[=value]*

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_I4

Remarks

The minimum value of the **Height** property is 460.

IgnoreCommunication

The **IgnoreCommunication** property sets or returns the value of the ignore cummunication option of NeuroCheck. If set to `TRUE`, NeuroCheck will ignore all functions requiring signals to be received via digital I/O or field bus; The default is `FALSE`.

Syntax

*object.***IgnoreCommunication** *[=value]*

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_BOOL

InterfaceVersion

The **InterfaceVersion** property returns the version number of the OLE automation interface (always greater or equal to 1).

Syntax

*object.***InterfaceVersion**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_I2

LastError

The **LastError** property returns a value indicating the most recent error that occured in NeuroCheck. Reading **LastError** resets the property to its initial value, i.e. to zero. The error codes are listed in the Quick Reference section of this manual.

Syntax

*object.***LastError**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_I4

Remarks

The **LastError** property encodes both a general error type and a detailed error number. With `ErrorType` denoting the general error type and `ErrorDetail` denoting the detailed error number, **LastError** is computed as follows:

```
   LastError = (ErrorType * 256) + ErrorDetail
```

In effect, the least significant byte of the return value contains the `ErrorDetail` number, one byte higher you will find the `ErrorType` value.

Example

The following Visual Basic® sample code reads the **LastError** property and returns the corresponding type and detail numbers (for the non-Visual-Basic programmers: the `"\"` denotes an integer division, `Mod` a modulo operation).

```
' ...
' do critical operation here, e.g. opening a check routine
' ...
' get debug information
Dim LastErrorNumber as Integer
LastErrorNumber = NCApplication.LastError
If LastErrorNumber <> 0 Then
    MsgBox "LastError returned " & CStr(LastErrorNumber) _
        & ", Error Type: "   & CStr(LastErrorNumber \ 256) _
        & ", Error Detail: " & CStr(LastErrorNumber Mod 256)
End If
' ...
```

Left

The **Left** property returns or sets the distance between the left edge of the physical screen and the main application window. See Figure 17.

Syntax

*object.***Left** *[=value]*

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_I4

LicenseLevel

The **LicenseLevel** property returns the license level of NeuroCheck encoded in the security key.

Syntax

*object.***LicenseLevel**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_I2

Possible result values of **LicenseLevel** are:

Constant	Value	Description
`NC_VERSION_LITE`	0	Demo version
`NC_VERSION_RUNTIME`	2	Runtime version
`NC_VERSION_FULL`	4	Premium Edition (fully licensed version)
`NC_VERSION_PROFESSIONAL`	16	Professional Edition

LicenseNumber

The **LicenseNumber** property returns the license number encoded in the security key.

Syntax

*object.***LicenseNumber**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_I4

Name

The **Name** property returns the name of the application.

Syntax

*object.***Name**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_BSTR

OperatingMode

The **OperatingMode** property returns or sets the operating mode of the application.

Syntax

*object.***OperatingMode** *[=value]*

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_I2

Possible settings for **OperatingMode** are:

Constant	Value	Description
`NC_MODE_MANUAL`	0	Manual mode (not available for runtime version)
`NC_MODE_TEST`	1	Test mode (read only)
`NC_MODE_LIVE`	2	Live mode
`NC_MODE_AUTOMATIC`	3	Automatic mode
`NC_MODE_AUTOCONFIG`	4	Configure automatic screen mode (read only)

Remarks

Changing the **OperatingMode** property requires a check routine to be loaded. Changing to Manual Mode requires an installation of Internet Explorer 4.0 or higher. Furthermore, Manual mode is not accessible for a runtime license of NeuroCheck. Both Test mode and Configure automatic screen mode are not accessible through OLE automation at all, i.e. setting the **OperatingMode** Property to one of the corresponding values will be without effect.

Parent

The **Parent** property returns NULL.

Syntax

*object.***Parent**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_DISPATCH

Path

The **Path** property returns the path specification for the executable file of the application.

Syntax

*object.***Path**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_BSTR

ReadFromBitmap

The **ReadFromBitmap** property sets or returns the simulate image capture option of NeuroCheck. If set to `TRUE`, all Transfer Image functions in the check routine will be switched to bitmap file instead of camera images; The default is `FALSE`.

Syntax

*object.***ReadFromBitmap** *[=value]*

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_BOOL

SubDeviceCount

The **SubDeviceCount** property returns the number of sub devices of the type specified by *DeviceType, DeviceIndex* and *SubDeviceType* configured in NeuroCheck's Device Manager. If the input arguments are incorrect, the property returns -1.

Syntax

*object.***SubDeviceCount(***DeviceType, DeviceIndex, SubDeviceType***)**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_I2

Argument	**Type**	**Description**
DeviceType	VT_I2	Type of the parent device.
DeviceIndex	VT_I2	Index of the parent device.
SubDeviceType	VT_I2	Type of the sub device.

For the possible settings for *DeviceType* please refer to the **DeviceCount** property of the **NCApplication** object. The possible settings for *SubDeviceType* are:

Constant	Value	Description
`NC_SUBDEVICE_CAMERA`	0	Camera
`NC_SUBDEVICE_DIGINPUT`	1	Digital input or field bus input bit
`NC_SUBDEVICE_DIGOUTPUT`	2	Digital output or field bus output bit

Remarks

Please note that the camera count indicates simply the number of camera channels for the given frame grabber. It does not correspond to the number of cameras currently activated in NeuroCheck's Device Manager.

SubDeviceName

The **SubDeviceName** property returns the name of a sub device configured in NeuroCheck's Device Manager. The device is specified by the input arguments *DeviceType, DeviceIndex, SubDeviceType* and *SubDeviceIndex.* If the input arguments are incorrect, the property returns an empty string.

Syntax

*object.***SubDeviceName(***DeviceType, DeviceIndex, SubDeviceType, SubDeviceIndex***)**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_BSTR

Argument	Type	Description
DeviceType	VT_I2	Type of the parent device.
DeviceIndex	VT_I2	Index of the parent device (counted from 0).
SubDeviceType	VT_I2	Type of the sub device.
SubDeviceIndex	VT_I2	Index of the sub device (counted from 0).

For the possible settings of *DeviceType* and *SubDeviceType* please refer to the **DeviceCount** and **SubDeviceCount** of the **NCApplication** object.

Top

The **Top** property returns or sets the distance between the top edge of the physical screen and the main application window. See Figure 17.

Syntax

*object.***Top** *[=value]*

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_I4

Width

The **Width** property returns or sets the distance between the left and right edges of the main application window. See Figure 17.

Syntax

*object.***Width** *[=value]*

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_I4

Remarks

The minimum value of the **Width** property is 620.

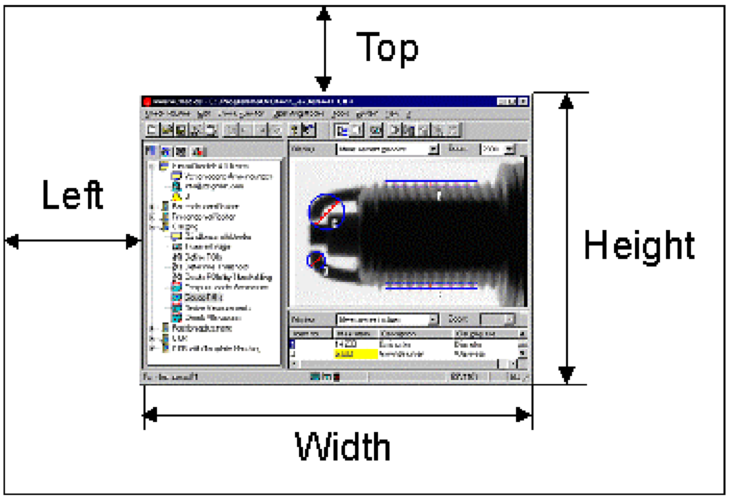


Figure 17: Properties for controlling size and position of the main application window.

WindowState

The **WindowState** property returns or sets a value indicating the visual state of the main application window at run time. The property returns -1 if it is not accessible . Using this property requires a check routine to be loaded.

Syntax

*object.***WindowState** *[=value]*

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_I2

Possible settings for **WindowState** are:

Constant	Value	Description
`NC_WINDOW_NORMAL`	1	Normal (Default)
`NC_WINDOW_MINIMIZED`	2	Minimized (minimized to an icon)
`NC_WINDOW_MAXIMIZED`	3	Maximized (enlarged to maximum size)

Remarks

As long as there is no check routine loaded in NeuroCheck, **WindowState** is not accessible and returns -1. Setting the **Visible** property of the **CheckRoutine** object to `TRUE` resets **WindowState** to its default value, i.e. to `NC_WINDOW_NORMAL`.

7.3.2 Methods

This section lists the methods available for the **NCApplication** object.

Execute

The **Execute** method starts execution of the active check routine in automatic mode. It does not take parameters. If the check routine was started successfully, the method returns `TRUE`. If an error occurred or if NeuroCheck is not operating in automatic mode, it returns `FALSE`.

Syntax

*object.***Execute()**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_BOOL

Remarks

Please note that the **Execute** method will not work

- for the demo version of NeuroCheck,
- if no check routine is loaded,
- if NeuroCheck is not configured for OLE control (under **System/Remote Control**) ,
- if NeuroCheck is not operating in automatic mode.

Examine the **LastError** property to find out the reason for the failure.

Open

The **Open** method opens an existing check routine. It takes the name of the check routine to be opened as argument and returns `TRUE`, if the file was opened successfully, or `FALSE`, if the check routine could not be opened.

Syntax

*object.***Open(***FileName***)**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_BOOL

Argument	Type	Description
FileName	VT_BSTR	Name of the check routine to be opened.

Remarks

Remember to release the object variable assigned to a check routine before opening a new check routine, as explained for the **ActiveCheckRoutine** property of the **NCApplication** object.

Quit

The **Quit** method closes the open check routine and exits the application.

Syntax

*object.*__Quit()__

The *object* placeholder represents the **NCApplication** object.

Return Type

None

ReadDigitalInput

The **ReadDigitalInput** method reads the current state of a digital input. It takes two arguments which specify the digital I/O board and the input to be read. The method returns the state of the input, i.e. `TRUE` or `FALSE`, if the input was read successfully, or VT_EMPTY if an error occurred.

Syntax

*object.*__ReadDigitalInput(__*BoardIndex, InputNumber*__)__

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_VARIANT

Argument	**Type**	**Description**
BoardIndex	VT_I2	Index of digital I/O board (counted from 0).
InputNumber	VT_I2	Number of the input to be read (counted from 0).

ReadDigitalInputWord

The **ReadDigitalInputWord** method reads the current states of all 16 digital inputs of one digital I/O board. It takes one argument which specifies the digital I/O board to be read. The method returns an integer value encoding the states of the inputs, or VT_EMPTY if an error occurred.

Syntax

*object.*__ReadDigitalInputWord(__*BoardIndex*__)__

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_VARIANT

Argument	Type	Description
BoardIndex	VT_I2	Index of digital I/O board (counted from 0).

Remarks

The return value of method **ReadDigitalInputWord** is the decimal value of a binary number encoding the current state of the 16 digital inputs. Bit *j* corresponds to input number *j*. A bit value of 1 indicates that the input is set, a value of 0 that the input is not set. For example, a return value of 9_{dec} = $(0000\ 0000\ 0000\ 1001)_{bin}$ means that only the inputs number 0 and number 3 are set.

ReadDigitalOutput

The **ReadDigitalOutput** method reads the current state of a digital output. It takes two arguments which specify the digital I/O board and the output to be read. The method returns the state of the output, i.e. `TRUE` or `FALSE,` if the output was read successfully, or VT_EMPTY if an error occurred.

Syntax

*object.***ReadDigitalOutput(***BoardIndex, OutputNumber***)**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_VARIANT

Argument	Type	Description
BoardIndex	VT_I2	Index of digital I/O board (counted from 0)
OutputNumber	VT_I2	Number of the output to be read (counted from 0)

ReadDigitalOutputWord

The **ReadDigitalOutputWord** method reads the current state of all 16 digital outputs of one digital I/O board. It takes one argument which specifies the digital I/O board to be read. The method returns an integer value encoding the state of the outputs, or VT_EMPTY if an error occurred.

Syntax

*object.***ReadDigitalOutputWord(***BoardIndex***)**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_VARIANT

Argument	Type	Description
BoardIndex	VT_I2	Index of digital I/O board (counted from 0).

Remarks

Please refer to the **ReadDigitalInputWord** methof for a description of the encoding of the return value.

ReadFieldBusInputBit

The **ReadFieldBusInputBit** method reads the current state of an input bit of a field bus device. It takes two arguments which specify the field bus board and the input bit to be read. The method returns the state of the input bit, i.e. `TRUE` or `FALSE,` if the input was read successfully, or VT_EMPTY if an error occurred.

Syntax

*object.***ReadFieldBusInputBit(***BoardIndex, InputNumber***)**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_VARIANT

Argument	Type	Description
BoardIndex	VT_I2	Index of field bus board (counted from 0).
InputNumber	VT_I2	Number of the input bit to be read (counted from 0).

ReadFieldBusOutputBit

The **ReadFieldBusOutputBit** method reads the current state of an output bit of a field bus device. It takes two arguments specifying the field bus board and the output bit to be read. The method returns the state of the output bit, i.e. `TRUE` or `FALSE,` if the output was read successfully, or VT_EMPTY if an error occurred.

Syntax

*object.***ReadFieldBusOutputBit(***BoardIndex, OutputNumber***)**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_BOOL or VT_EMPTY

Argument	Type	Description
BoardIndex	VT_I2	Index of field bus board (counted from 0).
OutputNumber	VT_I2	Number of the output bit to be read (counted from 0).

SetDigitalOutput

The **SetDigitalOutput** method sets the state of a digital output. It takes three arguments specifying the digital I/O board, the output to be set, and the state the output will be set to. The method returns `TRUE` if the output could be set, or `FALSE` if an error occurred.

Syntax

*object.***SetDigitalOutput(***BoardIndex, OutputNumber, NewState***)**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_BOOL

Argument	Type	Description
BoardIndex	VT_I2	Index of digital I/O board (counted from 0).
OutputNumber	VT_I2	Number of the output to be set (counted from 0).
NewState	VT_BOOL	State to which the output will be set.

SetDigitalOutputWord

The **SetDigitalOutputWord** method sets all 16 digital outputs of a digital I/O board at once. It takes three arguments, the first one specifying the digital I/O board to be set, the second one specifying the bits to be set and the third one holding the values of the bits. The method returns `TRUE` if the outputs could be set, or `FALSE` if an error occurred.

Syntax

*object.***SetDigitalOutputWord(***BoardIndex, BitMask, BitStates* **)**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_VARIANT

Argument	Type	Description
BoardIndex	VT_I2	Index of digital I/O board (counted from 0).
BitMask	VT_I4	Integer encoding the bit mask specifying the outputs to be set.
BitStates	VT_I4	Integer encoding the states to which the outputs will be set.

Remarks

Each of the 16 bits of the arguments *BitMask* and *BitStates* represents one of the 16 digital outputs. *BitMask* is used to specify the outputs to be altered. If bit number *j* is set in *BitMask*, i.e. if it is 1, then the corresponding output *j* will be set or reset according to bit number *j* in *BitStates*, otherwise the current state of the output will not be changed ("don't care"). A set bit in *BitStates* causes the corresponding output to be set, a bit value of 0 causes it to be reset – provided that the corresponding bit in the *BitMask* argument is set.

Example

The following example will set output number 0 and number 3 and reset output number 1, but it will not alter any of the other 13 outputs. Therefore *BitMask* is $(0000\ 0000\ 0000\ 1011)_{bin}$ = $0x000B_{hex} = 11_{dec}$ and *BitStates* is $(0000\ 0000\ 0000\ 1001)_{bin} = 0x0009_{hex} = 9_{dec}$

```
' ...
If Not NCApplication.SetDigitalOutputWord(0, 11, 9) Then
     MsgBox "Could not set digital outputs"
End If
' ...
```

SetFieldBusOutputBit

The **SetFieldBusOutputBit** method sets an output bit of a field bus device. It takes three arguments, specifying the field bus board, the output bit to be set and the state the output bit will be set to. The method returns `TRUE` if the output bit was set successfully, or `FALSE` if an error occurred.

Syntax

*object.***SetFieldBusOutputBit(***BoardIndex, OutputNumber, NewState***)**

The *object* placeholder represents the **NCApplication** object.

Return Type

VT_BOOL

Argument	Type	Description
BoardIndex	VT_I2	Index of field bus board (counted from 0).
OutputNumber	VT_I2	Number of the output bit to be set (counted from 0).
NewState	VT_BOOL	State to which the output bit will be set.

7.3.3 Wrapper

The Visual C++ ClassWizard will create the following wrapper class from the type library:

```
class INCApplication : public COleDispatchDriver
```

7.4 CheckRoutine Object

7.4.1 Properties

This section lists the properties of the **CheckRoutine** object. All constants are listed in `ncauto.h.`

☞ **Note:** C/C++ programmers must access the properties as follows:

ptrObj→**Get***PropertyName***()** for reading a property

ptrObj→**Set***PropertyName***(***NewValue***)** for setting a property (not available for read only properties)

The *ptrObj* placeholder represents a pointer to the **NCApplication** object, *PropertyName* the name of the property and *NewValue* the new value the property is set to.

ActiveScreenLayout

The **ActiveScreenLayout** property sets or returns the index of the screen layout of the automatic screen.

Syntax

*object.***ActiveScreenLayout** *[=value]*

The *object* placeholder represents the **CheckRoutine** object.

Return Type
VT_I2

Remarks
The **ActiveScreenLayout** property can only be changed in automatic mode.

ActiveScreenLayoutName

The **ActiveScreenLayoutName** property returns the designation of the currently used screen layout of the automatic screen.

Syntax
*object.***ActiveScreenLayoutName**

The *object* placeholder represents the **CheckRoutine** object.

Return Type
VT_BSTR

Remarks
The **ActiveScreenLayoutName** property is only accessible in automatic mode.

Application

The **Application** property returns the application object.

Syntax
*object.***Application**

The *object* placeholder represents the **CheckRoutine** object.

Return Type
VT_DISPATCH

Author

The **Author** property returns or sets the information about the author of the check routine. See Figure 18.

Syntax

*object.***Author** *[=value]*

The *object* placeholder represents the **CheckRoutine** object.

Return Type

VT_BSTR

Comments

The **Comments** property returns or sets the additional description of the check routine. See Figure 18.

Syntax

*object.***Comments** *[=value]*

The *object* placeholder represents the **CheckRoutine** object.

Return Type

VT_BSTR

Count

The **Count** property returns the number of individual checks in the check routine.

Syntax

*object.***Count**

The *object* placeholder represents the **CheckRoutine** object.

Return Type

VT_I2

CurrentCheckResult

The **CurrentCheckResult** property returns the result of the most recent execution of the individual check specified by argument *SCIndex*.

Syntax

*object.***CurrentCheckResult(***SCIndex***)**

The *object* placeholder represents the **CheckRoutine** object.

Return Type

VT_BOOL

Argument	Type	Description
SCIndex	VT_I2	Number of the individual check, whose result is to be retrieved.

CurrentResult

The **CurrentResult** property returns the final result of the most recent execution of the complete check routine.

Syntax

*object.***CurrentResult**

The *object* placeholder represents the **CheckRoutine** object.

Return Type

VT_BOOL

FileName

The **FileName** property returns the filename of the check routine, not including path.

Syntax

*object.***FileName**

The *object* placeholder represents the **CheckRoutine** object.

Return Type

VT_ BSTR

FullName

The **FullName** property returns the fully qualified path of the check routine file.

Syntax

*object.***FullName**

The *object* placeholder represents the **CheckRoutine** object.

Return Type

VT_ BSTR

Heading

The **Heading** property returns or sets the user-defined name of the check routine (root node in the structure tree view). See Figure 18.

Syntax

*object.***Heading** *[=value]*

The *object* placeholder represents the **CheckRoutine** object.

Return Type

VT_ BSTR

Remarks

Do not confuse this with the **Name** property of the check routine.

Name

The **Name** property returns the filename of the check routine, not including path.

Syntax

*object.***Name**

The *object* placeholder represents the **CheckRoutine** object.

Return Type

VT_ BSTR

Remarks

The **Name** property is supported for compatibility reasons, use the **FileName** property instead.

OID

The **OID** property sets or returns the unique object identification number (OID).

Syntax

*object.***OID**

The *object* placeholder represents the **CheckRoutine** object.

Return Type

VT_I4

Parent

The **Parent** property returns the application object.

Syntax

*object.***Parent**

The *object* placeholder represents the **CheckRoutine** object.

Return Type

VT_DISPATCH

PartsCheckedNOk

The **PartsCheckedNOk** property returns the total number of parts checked with final result "Part not O.K." since the active check routine has been opened.

Syntax

*object.***PartsCheckedNOk**

The *object* placeholder represents the **CheckRoutine** object.

Return Type

VT_I4

PartsCheckedOk

The **PartsCheckedOk** property returns the total number of parts checked with final result "Part O.K." since the active check routine has been opened.

Syntax

*object.***PartsCheckedOk**

The *object* placeholder represents the **CheckRoutine** object.

Return Type

VT_I4

Path

The **Path** property returns the path specification for the check routine, not including the filename or filename extension.

Syntax

*object.***Path**

The *object* placeholder represents the **CheckRoutine** object.

Return Type

VT_ BSTR

Saved

The **Saved** property returns a Boolean value indicating whether the check routine has been saved after the most recent change: `TRUE` if the check routine has not been changed since it was last saved, `FALSE` if the check routine has not been saved since the most recent change.

Syntax

*object.***Saved**

The *object* placeholder represents the **CheckRoutine** object.

Return Type

VT_ BOOL

Visible

The **Visible** property sets or returns whether the check routine object (in effect: the NeuroCheck window) is visible to the user or hidden; The default is `FALSE`.

Syntax

*object.***Visible** *[=value]*

The *object* placeholder represents the **CheckRoutine** object.

Return Type

VT_ BOOL

Remarks

According to the automation interface guidelines the visibility state is a property of the main data object, *not* the application object. Therefore, the visibility of the main application window has to be set through the data object.

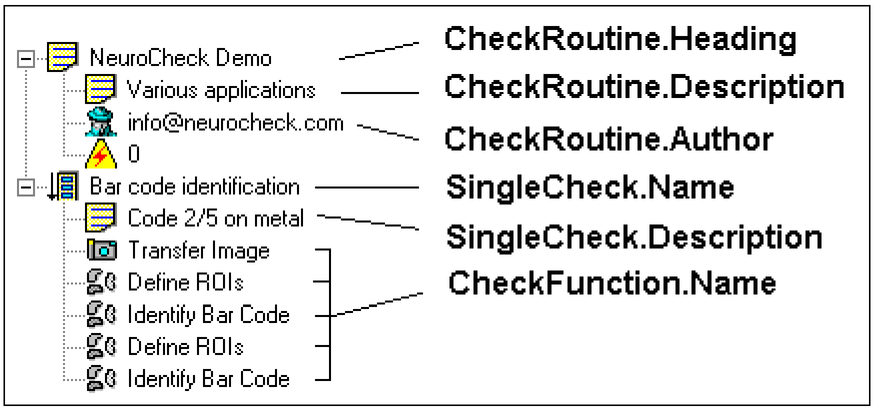


Figure 18: Descriptive properties of CheckRoutine, SingleCheck and CheckFunction object.

7.4.2 Methods

This section lists the methods of the **CheckRoutine** object.

Save

The **Save** method saves the check routine to the file specified in the **FullName** property of the check routine object. It returns a Boolean value indicating success or failure.

Syntax

*object.***Save**

The *object* placeholder represents the **CheckRoutine** object.

Return Type
VT_ BOOL

Remarks
The **Save** method fails if the check routine is protected by a password, i.e. it returns `FALSE`.

SaveAs

The **SaveAs** method saves changes to the check routine object in a different file specified by the *FileName* argument. It returns a Boolean value indicating success or failure.

Syntax
*object.***SaveAs(***FileName***)**

The *object* placeholder represents the **CheckRoutine** object.

Return Type
VT_ BOOL

Argument	Type	Description
FileName	VT_BSTR	Name of the new file, optionally including path.

Remarks
The **SaveAs** method does not verify whether the saving operation overwrites an existing file unless this file represents a check routine protected by a password. In this case, the check routine cannot be saved and **SaveAs** returns `FALSE`.

7.4.3 Collection Properties and Methods
The **CheckRoutine** object provides parts of the functionality of a collection object. Therefore one additional property and one additional method are available.

_NewEnum Property

The **_NewEnum** property provides an enumerator object that implements `IEnumVARIANT`. It is used by collection handling like the `For Each` loop in Visual Basic®.

Return Type

VT_DISPATCH

Example

The following Visual Basic® example displays the name of each individual check of the check routine in a message box.

```
' ...
Dim SingleCheck As Object
For Each SingleCheck In CheckRoutine
    MsgBox SingleCheck.Name
Next SingleCheck
' ...
```

Item Method

The **Item** method returns the given individual check object from the collection specified by argument *SCIndex.* If no check object is available for argument *SCIndex,* the **Item** method returns VT_EMPTY.

Syntax

*object.***Item(***SCIndex***)**

The *object* placeholder represents the **CheckRoutine** object.

Return Type

VT_DISPATCH

Argument	Type	Description
SCIndex	VT_VARIANT	Indicates the individual check object to be returned.

Remarks

If *SCIndex* is an integer value, the **Item** method returns the individual check object with position *SCIndex* in the collection of the **CheckRoutine** object. If *SCIndex* is a string, the **Item** method always returns the *first* individual check in the collection whose name is equal to string *SCIndex.*

Example

The following Visual Basic® sample code shows how to access the first individual check of the check routine's collection, and then the first individual check with name "OCR" (provided there exists a check with this name).

```
' ...
Dim SingleCheck As Object
Set SingleCheck = CheckRoutine.Item(0)
' ...
```

```
Set SingleCheck = CheckRoutine.Item(“OCR“)
If SingleCheck is Nothing Then
    MsgBox “Check ““OCR““ not found!“
End If
' ...
```

7.4.4 Wrapper

The Visual C++ ClassWizard will create the following wrapper class from the type library:

```
class ICheckRoutine : public COleDispatchDriver
```

7.5 SingleCheck Object

7.5.1 Properties

This section lists the properties of the **SingleCheck** object.

CheckEnabled

The **CheckEnabled** property sets or returns the current execution state of the check object. If `TRUE`, execution of the check is enabled, if `FALSE` it is disabled.

Syntax

*object.***CheckEnabled** *[=value]*

The *object* placeholder represents the **SingleCheck** object.

Return Type

VT_BOOL

Remarks

The **CheckEnabled** property replaces the methods **IsCheckEnabled** and **EnableCheck**.

Count

The **Count** property returns the number of check functions in the check.

Syntax

*object.***Count**

The *object* placeholder represents the **SingleCheck** object.

Return Type

VT_I2

CurrentResult

The **CurrentResult** property returns the final result of the most recent execution of the single check.

Syntax

*object.***CurrentResult**

The *object* placeholder represents the **SingleCheck** object.

Return Type

VT_BOOL

Description

The **Description** property returns or sets the description text of a check. See Figure 18.

Syntax

*object.***Description** *[=value]*

The *object* placeholder represents the **SingleCheck** object.

Return Type

VT_BSTR

LastFunction

The **LastFunction** property returns the index of the last check function whose result was "O.K." for the most recent execution of the individual check, i.e. the last check function that did not cause an "not O.K." either by a target value violation or because of an internal error.

Syntax

*object.***LastFunction**

The *object* placeholder represents the **SingleCheck** object.

Return Type

VT_I2

Remarks

LastFunction can be used to obtain information about which check function caused the "not O.K." of an individual check. If the **LastFunction** property returns -1, either the individual check has not been executed so far (**ErrorCode** property of first check function returns -1), or the first check function of the individual check returned "not O.K." (**ErrorCode** property greater or equal to 1).

Example

The following Visual Basic® code excerpt shows how to examine the cause of the failure of an individual check.

```
If Not SingleCheck.CurrentResult Then
    If SingleCheck.LastFunction < 0 And _
            SingleCheck.Item(0).ErrorCode < 0 Then
        MsgBox SingleCheck.Name & " was not executed"
    Else
        MsgBox SingleCheck.Item(SingleCheck.LastFunction).Name _
            & " failed"
    End If
End If
```

Name

The **Name** property returns or sets the user-defined name of a check. See Figure 18.

Syntax

*object.***Name** *[=value]*

The *object* placeholder represents the **SingleCheck** object.

Return Type

VT_BSTR

NumOfImages

The **NumOfImages** property returns the number of available images in the internal runtime data stack.

Syntax

*object.***NumOfImages**

The *object* placeholder represents the **SingleCheck** object.

Return Type

VT_I2

OID

The **OID** property sets or returns the unique object identification number (OID).

Syntax

*object.***OID**

The *object* placeholder represents the **SingleCheck** object.

Return Type

VT_I4

Parent

The **Parent** property returns the active **CheckRoutine** object.

Syntax

*object.***Parent**

The *object* placeholder represents the **SingleCheck** object.

Return Type

VT_DISPATCH

SCIndex

The **SCIndex** property returns the index of the individual check within the collection of the check routine.

Syntax

*object.***SCIndex**

The *object* placeholder represents the **SingleCheck** object.

Return Type

VT_I2

Example

The following sample code shows how to determine the current check result for the given individual check (of course the straight forward solution is to use the **CurrentResult** property of the **SingleCheck** object).

```
' ...
If CheckRoutine.CurrentCheckResult(SingleCheck.SCIndex) Then
    MsgBox SingleCheck.Name & " was OK."
End If
' ...
```

7.5.2 Methods

This section lists the methods of the **SingleCheck** object.

CopyImageToClipboard

Call the **CopyImageToClipboard** method to copy a gray level image from the internal runtime data stack to the Clipboard. The image to be copied is specified by the *ImageIndex* argument. The method returns a Boolean value indicating success or failure of the copy operation.

Syntax

*object.***CopyImageToClipboard(***ImageIndex, ImageFormat, ImageZoom***)**

The *object* placeholder represents the **SingleCheck** object.

Return Type

VT_BOOL

Argument	Type	Description
ImageIndex	VT_I2	Index of the image on the stack to be copied to the Clipboard.
ImageFormat	VT_I2	Format of the image copied to the Clipboard
ImageZoom	VT_I2	Scale factor of the image copied to the Clipboard.

Possible settings for *ImageFormat* are:

Constant	Value	Description
`NC_CF_DIB`	`8`	monochrome image in device-independent bitmap format
`NC_CF_COLOR_DIB`	`9`	color image in device-independent bitmap format

Possible settings for *ImageZoom* are:

Constant	Value	Description
`NC_ZOOM_200`	`200`	200 % of original image size
`NC_ZOOM_100`	`100`	100 % of original image size
`NC_ZOOM_50`	`50`	50 % of original image size
`NC_ZOOM_25`	`25`	25 % of original image size
`NC_ZOOM_10`	`10`	10 % of original image size

Example

The following example demonstrates how to copy the first image on the internal data stack to the clipboard. It is copied as a device-independent bitmap in normal size with color information. If no color information is present in the current image by default the normal grey-level image will be taken.

```
' ...
' Clear contents of clipboard
Clipboard.Clear
If SingleCheck.CopyImageToClipboard( _
     0, NC_CF_COLOR_DIB, NC_ZOOM_100) Then
     ' Copy image from Clipboard to Picture property of VB form
     Picture = Clipboard.GetData()
End If
' ...
```

EnableCheck

The **EnableCheck** method enables or disables execution of the individual check. It returns a Boolean value indicating the previous execution state.

Syntax

*object.***EnableCheck(***bEnable***)**

The *object* placeholder represents the **SingleCheck** object.

Return Type

VT_BOOL

Argument	Type	Description
bEnable	VT_BOOL	If `TRUE`, execution of the check will be enabled, if `FALSE` it will be disabled.

Remarks

Method **EnableCheck** is deprecated, please use property **CheckEnabled** instead.

GetImageData

The **GetImageData** method returns the gray level data of the image in the internal runtime data stack specified by argument *ImageIndex.* The method returns a safearray containing the gray level data of the image bytewise.

Syntax

*object.***GetImageData(***ImageIndex***)**

The *object* placeholder represents the **SingleCheck** object.

Return Type

VT_VARIANT

Argument	Type	Description
ImageIndex	VT_I2	Number of the image to be returned on the stack.

Remarks

This method is optimized for use with C/C++. See example in section 7.9.9.

GetImageProp

The **GetImageProp** method returns the properties of the image in the internal runtime data stack specified by the *ImageIndex* argument. The method returns a safearray containing the properties of the image.

Syntax

*object.***GetImageProp(***ImageIndex***)**

The *object* placeholder represents the **SingleCheck** object.

Return Type

VT_VARIANT

Argument	Type	Description
ImageIndex	VT_I2	Index of the image on the stack whose properties are to be returned.

Remarks

This method is optimized for use with C/C++. See example in section 7.9.9.

IsCheckEnabled

The **IsCheckEnabled** method returns the current execution state of the check object. If the return value is `TRUE`, execution of the check is currently enabled, if `FALSE` it is currently disabled.

Syntax

*object.***IsCheckEnabled**

The *object* placeholder represents the **SingleCheck** object.

Return Type

VT_BOOL

Remarks

Method **IsCheckEnabled** is deprecated, please use property **CheckEnabled** instead.

7.5.3 Collection Properties and Methods

The **SingleCheck** object provides parts of the functionality of a collection object. Therefore, one additional property and one additional method are available.

_NewEnum Property

The **_NewEnum** property provides an enumerator object that implements `IEnumVARIANT`. It is used by collection handling like the `For Each` loop in Visual Basic®.

Return Type

VT_DISPATCH

Example

The following example displays the name of each check function of the individual check in a message box.

```
' ...
Dim CheckFunction As Object
For Each CheckFunction In SingleCheck
    MsgBox CheckFunction.Name
Next CheckFunction
' ...
```

Item Method

The **Item** method returns the given check function object from the collection specified by argument *CFIndex.* If no check function object is available for argument *SCIndex,* the **Item** property returns VT_EMPTY.

Syntax

*object.***Item(***CFIndex***)**

The *object* placeholder represents the **SingleCheck** object.

Return Type

VT_DISPATCH

Argument	Type	Description
CFIndex	VT_VARIANT	Indicates the check function object to be returned.

Remarks

If *CFIndex* is an integer value, the **Item** method returns the check function object with position *CFIndex* in the collection of the **SingleCheck** object. If *CFIndex* is a string, the **Item** method always returns the *first* check function in the collection whose name is equal to string *CFIndex.*

Example

The following Visual Basic® sample code shows how to access the last check function of the individual check.

```
' ...
Dim CheckFunction As Object
Set CheckFunction = SingleCheck.Item(SingleCheck.Count-1)
' ...
```

7.5.4 Wrapper

The Visual C++ ClassWizard will create the following wrapper class from the type library:

```
class ISingleCheck : public COleDispatchDriver
```

7.6 CheckFunction object

7.6.1 Properties

This section lists the properties of the **CheckFunction** object.

Activated

The **Activated** property sets or returns a Boolean value indicating whether the check function is activated or deactivated.

Syntax

*object.***Activated** *[=value]*

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_BOOL

Remarks

The **Activated** property can only be changed for those check functions that allow deactivation. Currently, only check functions Filter Image and Determine Position can be deactivated. For all other check functions the **Activated** property returns `TRUE` always.

Category

The **Category** property returns the category to which the check function belongs. The category IDs are defined in `ncauto.h` and listed in the Quick Reference section of this manual.

Syntax

*object.***Category**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_I2

CFIndex

The **CFIndex** property returns the index of the check function within the collection of the individual check.

Syntax

*object.***CFIndex**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_I2

Remarks

Use the **CFIndex** property in combination with the **Parent** property and the **Item** method. The following sample code shows how to connect the check function object following the one currently stored in the `CheckFunction` variable with this object variable:

```
' ...
If CheckFunction.CFIndex < CheckFunction.Parent.Count Then
    Set CheckFunction = _
                CheckFunction.Parent.Item(CheckFunction.CFIndex + 1)
End If
' ...
```

ColsOfParameterMatrix

The **ColsOfParameterMatrix** property returns the number of columns of the check function's parameter matrix.

Syntax

*object.***ColsOfParameterMatrix**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_I2

ColsOfResultMatrix

The **ColsOfResultMatrix** property returns the number of columns of the check function’s result matrix.

Syntax

*object.***ColsOfResultMatrix**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_I2

ColsOfTargetValueMatrix

The **ColsOfTargetValueMatrix** property returns the number of columns of the check function’s target value matrix.

Syntax

*object.***ColsOfTargetValueMatrix**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_I2

ErrorCode

The **ErrorCode** property returns a value indicating the status of the check function after its most recent execution in automatic mode.

Syntax

*object.***ErrorCode**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_I2

The possible return values of **ErrorCode** are:

Constant	Value	Description
`NC_EXE_NOT`	`-1`	"Not Executed", either because the individual check has not been executed since opening the check routine or because a previous check function returned "not O.K." thus terminating execution of the individual check.
`NC_EXE_OK`	`0`	"O.K."
`NC_EXE_NOK`	`1`	"not O.K." because of a target value violation, e.g. a measurement did not conform to the tolerances or a bar code contained the wrong string.
`NC_EXE_ERROR`	`2`	"not O.K." because of an internal error; such an error occurs for example when an object required for computing a measurement is not present or a bar code is unreadable.

FunctionId

The **FunctionId** property returns the check function's internal function identification number. Refer to the Quick Reference section for a list of identification numbers.

Syntax

*object.***FunctionId**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_I2

Name

The **Name** property returns or sets the check function's (possibly user-defined) name. See Figure 18.

Syntax

*object.***Name** *[=value]*

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_BSTR

OID

The **OID** property sets or returns the unique object identification number (OID).

Syntax

*object.***OID**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_I4

Parent

The **Parent** property returns the **SingleCheck** object the current check function belongs to.

Syntax

*object.***Parent**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_DISPATCH

RowsOfParameterMatrix

The **RowsOfParameterMatrix** property returns the number of rows of the check function's parameter matrix.

Syntax

*object.***RowsOfParameterMatrix**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_I2

RowsOfResultMatrix

The **RowsOfResultMatrix** property returns the number of rows of the check function's result matrix.

Syntax

*object.***RowsOfResultMatrix**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_I2

RowsOfTargetValueMatrix

The **RowsOfTargetValueMatrix** property returns the number of rows of the check function's target value matrix.

Syntax

*object.***RowsOfTargetValueMatrix**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_I2

7.6.2 Methods

This section lists the methods of the **CheckFunction** object.

GetCurrentResult

The **GetCurrentResult** method returns the result values of the check function. It is only available after at least one execution in automatic mode. The method returns a safearray containing the current result values.

Syntax

*object.***GetCurrentResult**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_VARIANT

Remarks

This method is optimized for use with C/C++. See the explanation in section 7.7.1 and the example in section 7.9.8.

GetParameterItem

The **GetParameterItem** method returns an element of the check function’s parameter matrix. The element to be read is specified by the arguments *Row* and *Column*. The method returns the value of the element or VT_EMPTY on failure.

Syntax

*object.***GetParameterItem(***Row, Column***)**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_ VARIANT

Argument	Type	Description
Row	VT_I2	Row index of element in parameter matrix.
Column	VT_I2	Column index of element in parameter matrix.

Remarks

See the explanation in section 7.7.1.

GetParameters

The **GetParameters** method returns the current parameter settings of the check function. The method returns a safearray containing the current parameter settings.

Syntax

*object.***GetParameters**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_VARIANT

Remarks

This method is optimized for use with C/C++. See the explanation in 7.7.1and the example in section 7.9.6.

GetResultItem

The **GetResultItem** method returns an element of the check function's current result matrix. It is only available after at least one execution in automatic mode. The element to be read is specified by the arguments *Row* and *Column*. The method returns the value of the element, or VT_EMPTY on failure.

Syntax

*object.***GetResultItem(***Row, Column***)**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_ VARIANT

Argument	**Type**	**Description**
Row	VT_I2	Row index of element in result matrix.
Column	VT_I2	Column index of element in result matrix.

Remarks

See the explanation in section 7.7.1.

GetTargetValueItem

The **GetTargetValueItem** method returns an element of the check function's target value matrix. The element to be read is specified by the arguments *Row* and *Column*. The method returns the value of the element, or VT_EMPTY on failure.

Syntax

*object.***GetTargetValueItem(***Row, Column***)**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_ VARIANT

Argument	Type	Description
Row	VT_I2	Row index of element in target value matrix.
Column	VT_I2	Column index of element in target value matrix.

Remarks

See the explanation in section 7.7.1.

GetTargetValues

The **GetTargetValues** method returns the current target value settings of the check function. The method returns a safearray containing the current target value settings.

Syntax

*object.***GetTargetValues**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_VARIANT

Remarks

This method is optimized for use with C/C++. See the explanation in 7.7.1 and an example in section 7.9.7.

HasCurrentResult

The **HasCurrentResult** method returns a Boolean value giving information about the check function's ability to provide result values to be accessed by a controller application. If `TRUE`, the check function provides result values readable by a controller application, if `FALSE` it does not.

Syntax

*object.***HasCurrentResult**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_BOOL

Remarks

A return value of `TRUE` does not imply the existence of currently valid result values. Use the **ErrorCode** property to confirm that the result values are current.

```
' ...
If CheckFunction.ErrorCode < 0 Then
    MsgBox "Check function not executed!"
ElseIf CheckFunction.ErrorCode > 1 Then
    MsgBox "Check function aborted due to failure!"
ElseIf not CheckFunction.HasCurrentResult then
    MsgBox "Check function does not provide result values!"
Else
    ' Result is valid, read result
    ' ...
End If
' ...
```

HasParameters

The **HasParameters** method returns a Boolean value indicating the existence of parameter values which may be altered by a controller application. If `TRUE`, the check function has parameter settings which may be altered by a controller application, if `FALSE` it does not.

Syntax

*object.***HasParameters**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_BOOL

HasTargetValues

The **HasTargetValues** method indicates the existence of target values which may be altered by a controller application. If `TRUE`, the check function has target values which may be altered by a controller application, if `FALSE` it does not.

Syntax

*object.***HasTargetValues**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_BOOL

SetParameterItem

The **SetParameterItem** method sets an element of the check function’s parameter matrix. The element to be set is specified by the arguments *Row* and *Column*, its value by the argument *NewValue*. The method returns a Boolean value indicating success or failure of operation.

Syntax

*object.***SetParameterItem(***Row, Column, NewValue***)**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_ BOOL

Argument	**Type**	**Description**
Row	VT_I2	Row index of element in parameter matrix.
Column	VT_I2	Column index of element in parameter matrix.
NewValue	VT_VARIANT	New value of specified element.

Remarks

See the explanation in section 7.7.1.

SetParameters

The **SetParameters** method sets the parameter information of the check function. The parameter settings are passed in a safearray structure equivalent to the result value of method **GetParameters**. The method returns a value indicating success or failure of the operation.

Syntax

*object.***SetParameters(***ParaSetting***)**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_ BOOL

Argument	Type	Description
ParaSetting	VT_VARIANT	Safearray containing the parameter settings.

Remarks

This method is optimized for use with C/C++. See the explanation in section 7.7.1 and an example in section 7.9.6.

SetTargetValueItem

The **SetTargetValueItem** method sets an element of the check function's target value matrix. The element to be set is specified by the arguments *Row* and *Column*, its value by argument *NewValue*. The method returns a Boolean value indicating success or failure of the operation.

Syntax

*object.***SetTargetValueItem(***Row, Column, NewValue***)**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_ BOOL

Argument	Type	Description
Row	VT_I2	Row index of element in target value matrix.
Column	VT_I2	Column index of of element in target value matrix.
NewValue	VT_VARIANT	New value of specified element.

Remarks

See the explanation in section 7.7.1.

SetTargetValues

The **SetTargetValues** method sets the target value information of the check function. The target value settings are passed in a safearray structure equivalent to the result value of method **GetTargetValues**. The method returns a value indicating success or failure of the operation.

Syntax

*object.***SetTargetValues(***TargetSetting***)**

The *object* placeholder represents the **CheckFunction** object.

Return Type

VT_ BOOL

Argument	Type	Description
TargetSetting	VT_VARIANT	Safearray containing the parameter settings.

Remarks

This method is optimized for use with C/C++. See the explanation in section 7.7.1and an example in section 7.9.7.

7.6.3 Wrapper

The Visual C++ ClassWizard will create the following wrapper class from the type library:

```
class ICheckFunction : public COleDispatchDriver
```

7.7 Check Functions with Additional Automation Functionality

The following table lists the check functions providing additional functionality. The functionality is explained in detail in subsequent sections.

Function name	ID	Parameters	Target values	Result data
Identify Barcode	BCI=536	X	X	X
Identify DataMatrix Code	DMCI=552	X	X	X
Count ROIS	OBC=510		X	X
Evaluate Classes	CLE=543		X	X*
Check Allowances	GCHK=527		X*	X*
Copy ROIs	OCPY=525			X*
Determine Position	POSC=521			X
Capture Image	DIG=517	X		
Transfer Image	IDT=506	X		
Determine Threshold	ITH=508	X		
Define ROIs	DAOI=512	X		

Function name	ID	Parameters	Target values	Result data
Classify ROIs	`OCL=518`	X		
Screen ROIs	`OBF=513`	X*		
Rotate Image	`IROT=507`	X		
Template Matching	`TMA=546`	X		
Plug-In Functions		X		

[*] dynamic size of data: exchange through array of structures specific to the respective function

7.7.1 Data Exchange

The OLE automation interface of NeuroCheck offers two different ways to perform transfer of parameter, target value or result data for the check functions listed above.

Data Exchange by Element

Data exchange by element means that parameters, target value settings and current result values associated with a **CheckFunction** object are treated as a matrix of values. The appropriate size of each matrix is given by the function’s **RowsOf*xxx*Matrix** and **ColsOf*xxx*Matrix** properties, where ***xxx*** stands for **Parameter**, **TargetValue** or **Result**. Each element of the matrix can be passed in a variable of type VT_VARIANT to the function’s **Get*xxx*Item()** and **Set*xxx*Item()** methods. Row and column index are passed as input arguments to specify the data element to be read or set. The meanings of the elements for each check function type are explained in the following sections. In general, columns always contain the same type of information whereas the number of rows may be dynamic, indicated by [*] in the overview table, and by row index *n* in the explanation below. This method is more convenient for Visual Basic® programmers than using safearrays. It leads to optimal performance if only single elements of the data matrix need to be accessed.

Data Exchange by SafeArray

This method always passes the complete set of parameters, target values or result values in a safearray structure or an array of safearray structures using the **Get*yyy* ()** and **Set*yyy* ()** methods of the **CheckFunction** object, ***yyy*** standing for **Parameters**, **TargetValues** or **CurrentResult**. The pertaining structures for the different check function types are defined in `Ncauto.h` and explained below. This method minimizes the number of OLE automation calls and is best suited to pointer languages like C/C++ and to situations where the whole set of data needs to be transferred.

7.7.2 Identify Bar Code

Function Identify Bar Code allows parameter and target value setting and result data retrieval through OLE automation. The pertaining structures are defined in `Ncauto.h`.

Parameter	Item	Type	Description
Code type	(0,0)	VT_I4	Type of bar code. Possible settings are defined in `Ncauto.h`.
Line distance	(0,1)	VT_I4	Distance of search rays for scanning ROI [1,99].
Smoothing	(0,2)	VT_I4	Number of lines on both sides of search rays to be averaged [1,99].
Scan direction	(0,3)	VT_I4	Direction for scanning ROI. Possible settings are defined in `Ncauto.h`.
Check sum	(0,4)	VT_BOOL	If `TRUE`, the function performs a check sum test on the bar code. Only available for certain code types.
Characters	(0,5)	VT_I4	Number of characters contained in the code [1,99]. Only available for certain code types.

Target Value	Item	Type	Description
Check target code	(0,0)	VT_BOOL	If `TRUE`, the identified bar code is compared to the target string.
Target string	(0,1)	VT_BSTR	Bar code string to be present on the part.

Result Value	Item	Type	Description
Bar code string	(0,0)	VT_BSTR	String representing the identified bar code.

7.7.3 Identify DataMatrix Code

Function Identify DataMatrix Code allows parameter and target value setting and result data retrieval through OLE automation. The pertaining structures are defined in `Ncauto.h`.

Parameters	Item	Type	Description
Code type	(0,0)	VT_I4	Type of code, e.g. 12*12. Possible settings are defined in `Ncauto.h`.
Code color	(0,1)	VT_I4	Color of code, dark or light [0,1]. See also defines in `Ncauto.h`.
Code quality	(0,2)	VT_I4	Quality of code, good or poor [0,1]. See also defines in `Ncauto.h`.

Code size	(0,3)	VT_I4	Approximate code size in pixels [30,999].
Reference	(0,4)	VT_I4	Reference angle. Possible settings are defined in `Ncauto.h`.
Range	(0,5)	VT_I4	Range of angle [0-180°].
Undersampling	(0,6)	VT_I4	Sub sampling parameter [1-9].
Min. edge height	(0,7)	VT_I4	Contrast required for edges [10, 255].

Target Value	**Item**	**Type**	**Description**
Check target code	(0,0)	VT_BOOL	If `TRUE`, identified bar code is compared to target string.
Target string	(0,1)	VT_BSTR	DataMatrix code string to be present on the part.

Result Value	**Item**	**Type**	**Description**
DataMatrix code string	(0,0)	VT_BSTR	String representing the identified DataMatrix code.

7.7.4 Count ROIs

Function Count ROIs allows target value setting and result data retrieval through OLE automation. The pertaining structures are defined in `Ncauto.h`.

Target Value	**Item**	**Type**	**Description**
Check count	(0,0)	VT_BOOL	If `TRUE`, number of ROIs in first group is compared to target value.
Minimum	(0,1)	VT_I4	Minimum number of ROIs required in first group.
Maximum	(0,2)	VT_I4	Maximum number of ROIs allowed in first group.

Result Value	**Item**	**Type**	**Description**
Count	(0,0)	VT_I4	Current number of ROIs in first group.

7.7.5 Evaluate Classes

Function Evaluate Classes allows target value setting and result data retrieval through OLE automation. The pertaining structures are defined in `Ncauto.h`.

Target Value	**Item**	**Type**	**Description**
Verify	(0,0)	VT_BOOL	If `TRUE` the function will compare the classes

			of the ROIs in its input list to the class strings given.
Rejection threshold	(0,1)	VT_I4	An ROI will be classified as "Not O.K." if the classification certainty falls below this value, even if it corresponds to the respective class in the class list. (Valid range [0,100])
Class strings	(0,2-21)	VT_BSTR	Up to 20 class names to which the identified classes of the ROIs are to be compared.

The result values are returned in an array consisting of structures holding the following values for each identified ROI:

Result Value	Item	Type	Description
Class string	(*n*,0)	VT_BSTR	Name of the identified class.
Quality	(*n*,1)	VT_R4	Classification certainty.

7.7.6 Check Allowances

Function Check Allowances allows target value setting and result data retrieval through OLE automation. The pertaining structures are defined in `Ncauto.h`.
The target values are transferred using an array of structures holding the following values for each measurement:

Target Value	Item	Type	Description
Verify	(*n*,0)	VT_BOOL	If `TRUE`, the corresponding measurement will be compared to the pertaining target values.
Description	(*n*,1)	VT_BSTR	Name of the measurement. (read-only)
Nominal value	(*n*,2)	VT_R4	Nominal value of the measurement.
Lower allowance	(*n*,3)	VT_R4	Allowance added to the nominal value to compute the lower limit of the measurement, i.e. the requirement is: `current value ≥ nominal value + lower allowance`
Upper allowance	(*n*,4)	VT_R4	Allowance added to the nominal value to compute the upper limit of the measurement, i.e. the requirement is: `current value ≤ nominal value + upper allowance`

NOTE: to define a tolerance interval extending below the nominal value, the lower allowance has to be given with a negative sign.
The result values of the function are returned in an array of structures holding the following values for each measurement:

Result Value	Item	Type	Description

Description	(*n*,0)	VT_BSTR	Name of the measurement.
Current value	(*n*,1)	VT_R4	Current value of the measurement.

7.7.7 Copy ROIs

Function Copy ROIs allows result data retrieval through OLE automation. The pertaining structures are defined in `Ncauto.h`. The result values of the function are returned in an array of structures holding the following values for each feature of each ROI:

Result Value	Item	Type	Description
Object Number	(*n*,0)	VT_I4	Identifier of the ROI, counted from zero.
Feature ID	(*n*,1)	VT_I4	Identifier of the feature. The feature IDs are defined in `Ncauto.h` and listed in the Quick Reference section of this manual.
Current value	(*n*,2)	VT_R4	Current value of the feature.

NOTE: Function Copy ROIs is used to access the result data of function Compute Features. The reason why the results of Compute Features cannot be accessed directly is that this would increase execution time also for all users including those which do not use OLE automation at all. Therefore the less often used function Copy ROIs was chosen to give access to the feature values.
As a consequence, in order to read the result values of Compute Features through OLE automation, an instance of check function Copy ROIs has to be inserted after function Compute Features in the check routine.

7.7.8 Determine Position

Function Determine Position allows result data retrieval through OLE automation. The pertaining structures are defined in `Ncauto.h`.

Result Value	Item	Type	Description
X Offset	(0,0)	VT_R4	Offset in x direction of the reference object in the current image to the stored reference point.
Y Offset	(0,1)	VT_R4	Offset in y direction of reference object in the current image to the stored reference point.
Rotation	(0,2)	VT_R4	Rotation angle of the reference object in the current image relative to the stored orientation.
Pivot X	(0,3)	VT_R4	X coordinate of current pivot point.
Pivot Y	(0,4)	VT_R4	Y coordinate of current pivot point.

7.7.9 Capture Image

Function Capture Image allows parameter setting through OLE automation. The pertaining structure is defined in `Ncauto.h`. The elements of the parameter structure define the location of the image section to be transferred into memory.

Parameter	Item	Type	Description
Camera	(0,0)	VT_I4	Identifier (= index) of the camera the image is captured from.

7.7.10 Transfer Image

Function Transfer Image allows parameter setting through OLE automation. The pertaining structure is defined in `Ncauto.h`. The elements of the parameter structure define the location of the image section to be transferred into memory.

Parameter	Item	Type	Description
Left	(0,0)	VT_I4	X coordinate of top left corner.
Top	(0,1)	VT_I4	Y coordinate of top left corner.
Right	(0,2)	VT_I4	X coordinate of bottom right corner.
Bottom	(0,3)	VT_I4	Y coordinate of bottom right corner.
Source	(0,4)	VT_I4	Identifier of the image source. The source IDs are defined in `Ncauto.h` and listed below.
Camera	(0,5)	VT_I4	Identifier (= index) of the camera the image is captured from.
Bitmap	(0,6)	VT_BSTR	Name of bitmap file to be loaded. Not available if more than one bitmap file selected in Transfer Image.
Tray Index	(0,7)	VT_I4	Index of image tray the image

Possible settings for Source are:

Constant	Value	Description
`NC_IDT_SOURCE_CAMERA`	`0`	Image transferred from camera (i.e. frame grabber)
`NC_IDT_SOURCE_BITMAP`	`1`	Image loaded from bitmap file.
`NC_IDT_SOURCE_TRAY`	`2`	Image transferred from image tray.

7.7.11 Determine Threshold

Function Determine Threshold allows parameter setting through OLE automation. The pertaining structure is defined in `Ncauto.h`. The elements of the parameter structure define whether the threshold is computed manually or automatically, and certain parameters for each of the two cases.

Parameter	Item	Type	Description
Use manual threshold	(0,0)	VT_BOOL	If TRUE, the function uses the manual threshold, which can be set by OLE automation, else it performs automatic threshold computation.
Manual threshold	(0,1)	VT_I4	Value of the manual threshold to be used. (Valid range [0,255])
Result image	(0,2)	VT_I4	Parameter used for automatic threshold computation to adjust the predominance of light or dark areas in the result image. (Valid range [0,100])
Defect suppression	(0,3)	VT_I4	Parameter used for automatic threshold computation to suppress disturbances caused by very bright or very dark image regions. (Valid range [0,49])

7.7.12 Define ROIs

Function Define ROIs allows parameter setting through OLE automation. The pertaining structure is defined in `Ncauto.h`. The elements of the parameter structure define the location of the ROI defined first (ROI with ID 0) if this is a rectangular ROI.

Parameter	Item	Type	Description
Left	(0,0)	VT_I4	X coordinate of top left corner.
Top	(0,1)	VT_I4	Y coordinate of top left corner.
Right	(0,2)	VT_I4	X coordinate of bottom right corner.
Bottom	(0,3)	VT_I4	Y coordinate of bottom right corner.

7.7.13 Classify ROIs

Function Classify ROIs allows parameter setting through OLE automation. The pertaining structure is defined in `Ncauto.h`. The elements of the parameter structure define the file name of the classifier attached to this function.

Parameter	Item	Type	Description
Classifier	(0,0)	VT_BSTR	File name of classifier, possibly including path.

7.7.14 Screen ROIs

Function Screen ROIs allows parameter setting through OLE automation. The pertaining structure is defined in `Ncauto.h`.
The parameter values are transferred using an array of structures holding the following values for each feature currently available in function Screen ROIs:

Parameter	Item	Type	Description
Verify	(*n*,0)	VT_BOOL	If `TRUE`, the feature is activated for screening, else it is ignored.
Feature ID	(*n*,1)	VT_I4	Identifier of current feature. The feature IDs are defined in `Ncauto.h` and listed in the Quick Reference section of this manual.
Minimum	(*n*,2)	VT_R4	Minimum value allowed for the feature.
Maximum	(*n*,3)	VT_R4	Maximum value allowed for the feature.

NOTE: Group parameter settings of function Screen ROIs cannot be accessed through OLE automation. The feature ID is read-only, i.e. it cannot be changed. When setting the parameters in an array of structures, an arbitrary number of structures can be given. For instance, if four features are activated, but only the limits for one of them has to be changed, it is sufficient to only transfer the one structure element in the array that matches the feature ID.

7.7.15 Rotate Image

Function Rotate Image allows parameter setting through OLE automation. The pertaining structure is defined in `Ncauto.h`. The element of the parameter structure defines the operating mode of the check function.

Parameter	Item	Type	Description
Mode	(0,0)	VT_I4	Value indicating either the rotation angle for the rotation, or the direction of mirroring. Possible settings are defined in `Ncauto.h` and listed below.

Possible settings for Mode are:

Constant	Value	Description
`NC_IROT_ROTATE_0`	`0`	Rotation by 0 degrees counterclockwise
`NC_IROT_ROTATE_90`	`1`	Rotation by 90 degrees counterclockwise
`NC_IROT_ROTATE_180`	`2`	Rotation by 180 degrees counterclockwise
`NC_IROT_ROTATE_270`	`3`	Rotation by 270 degrees counterclockwise
`NC_IROT_MIRROR_HORIZONTAL`	`4`	Mirror image horizontally
`NC_IROT_MIRROR_VERTICAL`	`5`	Mirror image vertically

7.7.16 Template Matching

Function Template Matching allows parameter setting through OLE automation. The pertaining structure is defined in `Ncauto.h`. The element of the parameter structure defines the operating mode of the check function. The elements of the parameter structure apply ton of the first group of ROIs only.

Parameter	Item	Type	Description
Result Positions	(0,0)	VT_I4	Number of objects the function will create at most inside the available regions.
Minimum quality	(0,1)	VT_I4	Required degree of correspondence to the template an object must reach to be created as new region of interest [0,100].

7.8 Source Code Samples for Visual Basic®

The source code in this section was taken mostly verbatim from the official NeuroCheck OLE automation Controller Examples implemented in Visual Basic 6.0. You can find the projects including source code in the folder `\Programming\OLE` in the NeuroCheck installation path.

7.8.1 Attach and Detach Server, Open a Check Routine

The following source code sample demonstrates how to establish the connection to the automation server upon opening the form of the controller application, how to request information, open and access a check routine and change the visibility state of the NeuroCheck server. Also shown is how to detach the server upon closing of the controller application's main form.

```
Option Explicit

' OLE access objects on module level
Dim NeuroCheck      As Object   ' NCApplication object
Dim CheckRoutine    As Object   ' CheckRoutine object
' Note: if the NCheck.tlb type library is included to the VB project,
'       then the automation objects may also be defined as follows:
'       Dim NeuroCheck      As NCheck.Application
'       Dim CheckRoutine    As NCheck.CheckRoutine

' Constants specific to NeuroCheck
' Constants are defined in NcAuto.bas

Private Sub Form_Load()
On Error GoTo End_Application

    ' Create OLE server object (variable on module level)
    Set NeuroCheck = CreateObject("NeuroCheck.Application")
    If NeuroCheck Is Nothing Then
        MsgBox "Unable to launch server application!"
        GoTo End_Application
    End If

    ' At this point we are attached to server NeuroCheck (V4.x and higher)!
    ' Server process is running as "hidden" software component:
    ' there is no entry in the system's taskbar as long as the
    ' visibility state of the data object is set to "hidden" (the default).

    ' Now read server information
    If NeuroCheck.LicenseLevel = NC_VERSION_LITE Then
        MsgBox "You are running a demo version of NeuroCheck." & _
              "Execute will not work for the demo version!"
    End If

    ' Get file name of check routine to be opened
    Dim strFileName As String
    If Not GetCheckRoutineName(strFileName) Then
        MsgBox "Could not get check routine name!"
        GoTo End_Application
    End If

    ' Release check routine object before opening new one
    Set CheckRoutine = Nothing

    ' Open check routine in server
    If Not NeuroCheck.Open(strFileName) Then
        MsgBox "Could not open " & strFileName
        GoTo End_Application
    End If

    ' Get check routine object (object veriable on module level)
    Set CheckRoutine = NeuroCheck.ActiveCheckRoutine
    If CheckRoutine Is Nothing Then
        MsgBox "Could not access check routine object!"
        GoTo End_Application
    End If

    ' Switch to automatic mode using the "Operating Mode" property
    ' (only possible after check routine was opened successfully)
    NeuroCheck.OperatingMode = NC_MODE_AUTOMATIC

    ' Switch visibility state to visible
    ' (only possible after check routine was opened successfully)
    CheckRoutine.Visible = True

    ' Finally, enable Start button
    cmdStart.Enabled = True

    Exit Sub

End_Application:
    ' On error close application.
    ' NeuroCheck server will be detached upon unloading the form,
    ' i.e. in procedure Form_Unload()
    Unload Me

End Sub


Private Sub Form_Unload(Cancel As Integer)
On Error Resume Next

    ' upon unload event of form, close connection to server
    If Not NeuroCheck Is Nothing Then
        ' Detach NeuroCheck server by quit
        ' Note: changes to the check routine will NOT be saved.
        NeuroCheck.Quit
        Set NeuroCheck = Nothing
    End If

End Sub
```

7.8.2 Access Single Checks and Check Functions

The following source code sample shows how to access individual checks and check functions of the check routine, and how to read information. The first example shows that the collection property makes it possible to use a `for each` loop, the second example uses a "normal" `for` loop.

```
' ...
' Loop over all checks and display name, if check is not enabled
Dim SingleCheck As Object
For Each SingleCheck In CheckRoutine
     If Not SingleCheck.CheckEnabled Then
          MsgBox SingleCheck.Name & " is disabled"
     End If
Next SingleCheck

' Loop over all checks and display check that failed for most recent
' exeuction of check routine
Dim i As Integer
For i = 0 To CheckRoutine.Count - 1
     If Not CheckRoutine.CurrentCheckResult(i) Then
          MsgBox CheckRoutine.Item(i).Name & " failed!"
     End If
Next i

' Loop over all check functions of first single check
' and display check functions that provide target values
Dim CheckFunction As Object
For Each SingleCheck in CheckRoutine
     For Each CheckFunction in SingleCheck
          If CheckFunction.HasTargetValues then
               MsgBox CheckFunction.Name & " has target values!"
          End If
     Next CheckFunction
Next SingleCheck
```

7.8.3 Automatic Mode Execution

The following source code sample demonstrates how to execute the check routine in automatic mode and how to retrieve the final result of the check routine.

```
Private Sub cmdStart_Click()
On Error GoTo Err_cmdStart_Click

     ' Start execution of NeuroCheck
     If Not NeuroCheck.Execute Then
          ' Error occured on execution
          ' give acoustic warning and display message box
          Beep
          ' evaluate LastError property of NeuroCheck,
          ' done in separate function called GetLastErrorString
          MsgBox "Execute failed:" & vbCrLf & GetLastErrorString(NeuroCheck)
          GoTo Exit_cmdStart_Click
     End If

     ' Verify result of check routine
     If (CheckRoutine.CurrentResult) Then
          ' Result was OK
          ' ...
     Else
```

```
              ' Result was not OK
              ' ...
          End If

          ' Display identified barcode in Label control of form
          ' (see below ...)

          ' Display image in PictureBox control of form
          ' (see below ...)

          ' Display borders of ROI used to identify Barcode
          ' (see below ...)

     Exit_cmdStart_Click:
          Exit Sub

     Err_cmdStart_Click:
          MsgBox "Error " & Err & ": " & Error
          Resume Exit_cmdStart_Click

     End Sub
```

7.8.4 Check Function Result

The following source code excerpt demonstrates how to read out the result of a check function after execution in automatic mode. Function Identify Bar Code simply returns the identified bar code in element (0,0) of its result matrix.

```
' ...
' Assign object variable CheckFunction
' ...

' Verify check function
If CheckFunction.FunctionID <> NC_FCT_BCI Then Exit Function

' Verify error code of check function
If CheckFunction.ErrorCode < 0 Then
     ' check function was not executed
     labValue.Caption = "-----"
     GoTo Exit_DisplayValue
ElseIf CheckFunction.ErrorCode > 1 Then
     ' check function caused error due to internal failure
     labValue.Caption = "ERROR"
     GoTo Exit_DisplayValue
End If

' Read element (0,0) of result matrix
Dim varValue As Variant
varValue = CheckFunction.GetResultItem(0, 0)

' Verify VarType of varValue
If VarType(varValue) <> vbString Then
     ' VarType should be String for barcode!
     ' ...
Else
     ' Show result string in Label element
     labValue.Caption = CStr(varValue)
End If
```

7.8.5 Check Function Target Values

The following source code excerpt demonstrates how to change the target value settings of check function Check Allowances. The function sets the nominal value of each measurement to the most recent result of the function, i.e. it may be used to implement a "Teach" modus. Note that this functionality requires to access both the result value matrix and the target value matrix of the check function.

```
' ...
' Assign object variable CheckFunction
' ...

' Verify check function
If CheckFunction.FunctionID <> NC_FCT_GCHK Then Exit Function

Dim row As Integer
Dim strNameTarget  As String
Dim strNameResult As String
Dim varResultValue As Variant

With CheckFunction

    ' verify size of target and result matrix
    If .RowsOfTargetValueMatrix <> .RowsOfResultMatrix Then
        Exit Function
    End If

    For row = 0 To .RowsOfTargetValueMatrix - 1

        ' only change nominal value, if it is used in GCHK
        If .GetTargetValueItem(row, 0) Then

            ' compare names of measurement, just to be sure
            strNameTarget = .GetTargetValueItem(row, 1)
            strNameResult = .GetResultItem(row, 0)
            If StrComp(strNameTarget, strNameResult) <> 0 Then
                MsgBox "Names differ for row " & CStr(row)
            End If

            ' get result value
            varResultValue = .GetResultItem(row, 1)

            ' set target value
            If Not .SetTargetValueItem(row, 2, varResultValue) Then
                MsgBox "Could not set target value for " & strNameTarget
            End If
        End If

    Next row

End With
```

7.8.6 Display Image

The following source code excerpt shows how to copy a complete image in DIB (Device Independent Bitmap) format to the clipboard, and how to assign the contents of the clipboard to a PictureBox control of the controller form. It also shows how to read the parameter setting of check function Define ROIs in order to manually draw the corresponding ROI.

```
' ...
' Get index of image and zoom factor (or use constant NC_ZOOM_xxx)
' ...

' Clear clipboard
Clipboard.Clear
```

```
' Copy image of check to clipboard
If SingleCheck.CopyImageToClipboard(intImageIndex, _
                                    NC_CF_COLOR_DIB, intZoom) Then
    ' now assign contents of clipboard to Picture property
    ' of PictureBox control
    pBox.Picture = Clipboard.GetData()
End If

' Assign object variable for appropriate check function "Define ROI"
' ...

' Read paramters of check function "Define ROI"
Dim iLeft, iTop, iRight, iBottom
With CheckFunction
    iLeft = .GetParameterItem(0, 0) * intZoom / 100
    iTop = .GetParameterItem(0, 1) * intZoom / 100
    iRight = .GetParameterItem(0, 2) * intZoom / 100
    iBottom = .GetParameterItem(0, 3) * intZoom / 100
End With

' Draw the rectangular ROI
pBox.ScaleMode = vbPixels
pBox.Line (iLeft, iTop)-(iLeft, iBottom), RGB(0, 0, 255)
pBox.Line (iLeft, iTop)-(iRight, iTop), RGB(0, 0, 255)
pBox.Line (iRight, iBottom)-(iLeft, iBottom), RGB(0, 0, 255)
pBox.Line (iRight, iBottom)-(iRight, iTop), RGB(0, 0, 255)
```

7.8.7 Access Digital Inputs and Outputs

The following source code excerpt shows how to read a digital input, start execution and set a digital output depending on the execution's result.

```
Const BoardIndex  As Integer = 0  ' Index of digital IO board
Const InputIndex  As Integer = 0  ' Index of digital input to be read
Const OutputIndex As Integer = 1  ' Index of digital output to be set
Dim varInputValue As Variant      ' Value of digital input

' Verify, if digital IO board is configured in NeuroCheck
If NeuroCheck.DeviceCount(NC_DEVICE_DIGITALIO) <= BoardIndex Then
    Exit Function
End If

' Read first input of first digital IO board
varInputValue = NeuroCheck.ReadDigitalInput(BoardIndex, InputIndex)

' Verify type of input
If VarType(varInputValue) <> vbBoolean Then
    ' an error occured for reading the digital IO board
    MsgBox "Could not read digital input no. " & CStr(InputIndex)
End If

If varInputValue Then

    ' Execute check and get result
    Dim bCheckResult As Boolean
    If NeuroCheck.Execute Then
        bCheckResult = CheckRoutine.CurrentResult
    Else
        bCheckResult = False
    End If

    ' Set digital second output depending of check result
    If Not NeuroCheck.SetDigitalOutput( _
        BoardIndex, OutputIndex, bCheckResult) Then
```

```
            MsgBox "Could not set digital output no. " & CStr(OutputIndex)
        End If

    End If
```

7.9 Source Code Samples for C/C++

The source code in this section was taken mostly verbatim from the official NeuroCheck OLE automation Controller Example implemented in Visual C++. A simpler example is available in source code in the folder `\Programming\OLE\NcCppSimple` in the NeuroCheck installation path.

7.9.1 Contact Server

The following code sample demonstrates how to establish the connection to the automation server, request information about the server and detach the connection again.

```
void OnAttachServer()
{
    // create OleDispatchDriver object
    INCApplication* pINCApplication = new INCApplication();
    BOOL bConnected = FALSE;
    try
    {
        bConnected = pINCApplication->
            CreateDispatch("NeuroCheck.Application");
    }
    catch(...)
    {
        TRACE0("OLE exception during create of IDispatch\n");
    }
    if (FALSE == bSuccess)
    {
        delete pINCApplication;
        TRACE0("error: unable to launch server application\n");
        return;
    }

    // at this point we are attached to server NeuroCheck (V4.x ++)!
    // server process is running as a "hidden" software component:
    // there's no entry in the system's taskbar as long as the visibility
    // state of the data object is set to "hidden" (the default).

    // now read server information
    short int iMajorVersion = pINCApplication->GetExeMajorVersion();
    short int iMinorVersion = pINCApplication->GetExeMinorVersion();
    CString szPath          = pINCApplication->GetPath();
    long lLicenseNumber     = pINCApplication->GetLicenseNumber();
    short int iLicenseLevel = pINCApplication->GetLicenseLevel();
    switch(iLicenseLevel)
    {
    case NC_VERSION_RUNTIME:   // 0x02
        TRACE1("connected to runtime version no %ld\n",
               lLicenseNumber);
        break;
    case NC_VERSION_FULL: // 0x04
        TRACE1("connected to Premium Edition no %ld\n",
               lLicenseNumber);
        break;
        case NC_VERSION_PROFESSIONAL:  // 0x10
            TRACE1("connected to Professional Edition no %ld\n",
                   lLicenseNumber);
            break;
        default:
            break;
        }

        // quit server and detach again
        pINCApplication->Quit();
        delete pINCApplication;
        TRACE0("disconnected\n");
    } // eof function OnAttachServer()
```

7.9.2 Access Check Routine

The following source code sample demonstrates how to open a check routine and determine information about its individual checks, while the controller is attached to the server.

```
void OnOpenCheckRoutine()
{
    CString szPath;

    // do something here to get path information...

    // now open check routine in server
    if (pINCApplication->Open((LPCTSTR)szPath))
    {

        // create check routine object
        LPDISPATCH lpDis = pINCApplication->GetActiveCheckRoutine();
        ICheckRoutine* pICr = new ICheckRoutine(lpDis);

        // determine number of individual checks
        int iCount = pICr->GetCount();

        // loop over all checks and read their description texts
        ISingleCheck* pISc;
        for (int i = 0; i < iCount; i++)
        {

            // get individual check object
            COleVariant var ((short)i, VT_I2);
            LPDISPATCH lpDisCheck = pICr->Item(var);
            ASSERT(NULL != lpDisCheck);

            // create dispatch driver object
            pISc = new ISingleCheck(lpDisCheck);

            // read name
            CString szName = pISc->GetName();
            TRACE1("name = %s", (const char*)szName);
            if (FALSE == pISc->GetCheckEnabled())

            // modify execution state of check
            pISc->SetCheckEnabled(TRUE);

            // destroy dispatch driver object
            delete pISc;

        } // eofor

        delete pICr;

        } // eoif
```

```
          else
          {
               TRACE0("unable to open check routine!\n");
          }
     } // eof function OnOpenCheckRoutine()
```

7.9.3 Access Check Functions

The following source code sample demonstrates how to read the list of check functions of an individual check and how to display the help page for a particular check function.

```
// ID of check function "IDT","Transfer image"
// NC:FCT_IDT declared in ncauto.h

void ShowCheckDetails(ISingleCheck* pISc)
{
     // get number of check functions
     int iCount = pISc->GetCount();
     ICheckFunction* pICf;

     // loop over check functions
     for (int i = 0; i < iCount; i++)
     {
          // create check function object
          COleVariant var ((short)i, VT_I2);
          LPDISPATCH lpDis = pISc->Item(var);
          ASSERT(NULL != lpDis);

          // create dispatch driver object for check function
          pICf = new ICheckFunction(lpDis);

          // read name
          CString szName = pICf->GetName();
          TRACE1("name = %s", (const char*)szName);

          // display help page, if function is "Transfer image"
          if (NC_FCT_IDT == pICf->GetFunctionId())
          {
               // first retrieve help file path...
               CString szHelpFilePath;

               // After retrieving help file path,
               // call WinHelp with context information
               ::WinHelp(hwnd, (const char*)szHelpFilePath,
                         HELP_CONTEXT, (DWORD)NC_FCT_IDT);

          } // eoif

          delete pICf;

     } // eofor

} // eof function ShowCheckDetails
```

7.9.4 Automatic Mode Execution

The following source code sample demonstrates how to switch to automatic mode after establishing the connection to the server and loading a check routine, how to execute the check routine in automatic mode and retrieve the final result.

```
void OnStartAutomaticMode()
{

    // ...
    // switch to automatic mode using the
    // "OperatingMode" property of the application object
    pINCApplication->SetOperatingMode(NC_MODE_AUTOMATIC);

    // execute active check routine
    if (TRUE == pINCApplication->Execute())
    {

        // determine most recent result and statistics
        BOOL bResult = pICheckRoutine->GetCurrentResult();
        int iOk = pICheckRoutine ->GetPartsCheckedOk();
        int iNOk = pICheckRoutine ->GetPartsCheckedNOk();
        TRACE3(
            "result = %u [total no. of Ok = %d, Nok = %d]\n",
            (unsigned int)bResult, iOk, iNOk);

    } // eoif
    else
    {

        TRACE0("error: unable to execute check routine!\n");
    }

    // back to manual mode (not available for runtime license!)
    pINCApplication->SetOperatingMode(NC_MODE_MANUAL);

}
```

7.9.5 Display Live Image

The following source code sample demonstrates how to switch to live mode and make the server application visible after establishing the server connection and loading a check routine.

```
void OnLiveMode()
{

    // ...
    // switch to live image mode
    pINCApplication->SetOperatingMode(NC_MODE_LIVE);

    // remark: according to the automation interface guidelines
    // the visibility state is a property of the main data        //
object, NOT of the application object.
    // Therefore toggle the visibility state of the data object
    // "check routine"
    pICheckRoutine->SetVisible(TRUE);

} // eof function OnLiveMode()
```

7.9.6 Check Function Parameters

The following source code sample demonstrates how to read out the parameters of a check function, modify them and store them back to the check function object.

```
void ModifyParameters(ICheckFunction* pICf)
{

    if (FALSE == pICf->HasParameters())
```

```
{
    TRACE0("no parameter settings available!\n");
    return;
} // eoif

// read current parameter settings
VARIANT arg = pICf->GetParameters();

// error check
// data is transfered between server and controller using
// a safearray
if (arg.vt != (VT_UI1 | VT_ARRAY))
{
    TRACE0("error: invalid return value!\n");
    return;
} // eoif

switch(pICf->GetFunctionId())
{

case NC_FCT_IDT: // transfer image, define in ncauto.h
    {
        sNcIdtParameter* psPara; // declared in ncauto.h

        // access data
        if (S_OK != ::SafeArrayAccessData(
                arg.parray, (void**)&psPara))
        {
            TRACE0("error safe array access!\n");
            return;
        } // eoif

        // assign to local vars
        unsigned int uiLeft = psPara->uiLeft;
        unsigned int uiTop  = psPara->uiTop;

        // release access
        ::SafeArrayUnaccessData(arg.parray);

        // notify system that safearray is no longer needed
        ::SafeArrayDestroy(arg.parray);

        // ---------------------------------------------
        // ...do something with the data;
        // launch a dialog for instance and let the user
        // modify the data
        // ---------------------------------------------

        // fill local struct and transfer the data back
        // to the server
        sNcIdtParameter sPara;
        para.uiLeft = 100;
        para.uiTop = 200;
        SAFEARRAYBOUND rgb [] = { sizeof(sPara), 0 };

        // create a 1-dimensional safearray
        SAFEARRAY* psa = ::SafeArrayCreate(
                            VT_UI1, 1, rgb);
        if (psa)
        {
            void* pData;
            ::SafeArrayAccessData(psa, (void**)&pData);

            // serialize the data into the char array
```

```
                memcpy(pData, &sPara, rgb->cElements * sizeof(char));

                // release
                ::SafeArrayUnaccessData(psa);

                // now pack the data into the VARIANT
                VARIANT arg;
                VariantInit(&arg);
                arg.vt     = VT_UI1 | VT_ARRAY;
                arg.parray = psa;

                // transmit
                if (FALSE == pICf->SetParameters(arg))
                    TRACE0("error!\n");
            } // eoif(psa)
            break;
        } // eof case NC_FCT_IDT

    case NC_FCT_ITH: // determine threshold, define in ncauto.h
        {
            sNcIthParameter* psPara;       // declared in ncauto.h

            // ... corresponding to above case
            break;
        } // eof case NC_FCT_ITH

    default:
        break;
    } // eof switch

} // eof function ModifyParameters()
```

7.9.7 Check Function Target Values

The following source code sample demonstrates how to read out the target value settings of a check function, modify them and store them back to the check function object. Function Check ROIs uses a simple structure for its target values, function Check Allowances an array of target value structures.

```
void ModifyTargetValues(ICheckFunction* pICf)
{

    if (FALSE == pICf->HasTargetValues())
    {
        TRACE0("no target values available!\n");
        return;
    } // eoif

    // read current parameter settings
    VARIANT arg = pICf->GetTargetValues();

    // error check
    // data is transfered between server and controller using
    // a safearray
    if (arg.vt != (VT_UI1 | VT_ARRAY))
    {
        TRACE0("error: invalid return value!\n");
        return;
    } // eoif

    switch(pICf->GetFunctionId())
    {

    case NC_FCT_OBC: // Count ROIs, define in ncauto.h
```

```
        {
            sNcObcTargetValue* psTv;   // declared in ncauto.h

            // access data
            if (S_OK != ::SafeArrayAccessData(arg.parray, (void**)&psTv))
            {
                TRACE0("error safe array access!\n");
                return;
            } // eoif

            // assign to local vars
            BOOL bCheckData    = psTv->bCheckData;
            unsigned int uiMin = psTv->uiMin;
            unsigned int uiMax = psTv->uiMax;

            // release access
            ::SafeArrayUnaccessData(arg.parray);

            // notify system that safearray is no longer needed
            ::SafeArrayDestroy(arg.parray);

            // ---------------------------------------------
            // ...do something with the data;
            // launch a dialog for instance and let the user
            // modify the target values
            // ---------------------------------------------

            // fill local struct and transfer the data back
            // to the server
            sNcObcTargetValue sTv;
            sTv.uiMin = 17;
            sTv.uiMax = 21;
            SAFEARRAYBOUND rgb [] = { sizeof(sTv), 0 };

            // create a 1-dimensional safearray
            SAFEARRAY* psa = ::SafeArrayCreate(VT_UI1, 1, rgb);
            if (psa)
            {
                void* pData;
                ::SafeArrayAccessData(psa, (void**)&pData);

                // serialize the data into the char array
                memcpy(pData, &sTv, rgb->cElements * sizeof(char));

                // release
                ::SafeArrayUnaccessData(psa);

                // now pack the data into the VARIANT
                VARIANT arg;
                VariantInit(&arg);
                arg.vt     = VT_UI1 | VT_ARRAY;
                arg.parray = psa;

                // transmit
                if (FALSE == pICf->SetTargetValues(arg))
                    TRACE0("error!\n");
            } // eoif (psa)
        break;
    } // eof case NC_FCT_OBC

    case NC_FCT_GCHK: // Check Allowances, define in ncauto.h
        {
            // advanced functionality:
            // an array of structs is serialized into the safearray
```

```
SAFEARRAY* psa = arg.parray;
unsigned int uiNumOfElements =
     (psa->rgsabound[0].cElements /
         sizeof(sNcGchkTargetValue));
     TRACE1("%u struct elements\n", uiNumOfElements);
     if (0 == uiNumOfElements)
         return;

// access data
sNcGchkTargetValue* psTv;
::SafeArrayAccessData(arg.parray, (void**)&psTv);
for (unsigned int ui = 0; ui < uiNumOfElements; ui++)
{
     float fRefValue = psTv[ui].fRefValue;
     CString szName  = psTv[ui].szName;
     TRACE2("%s = %f\n", (const char*)szName, fRefValue);
}

// release access
::SafeArrayUnaccessData(arg.parray);

// notify system that safearray is no longer needed
::SafeArrayDestroy(arg.parray);

// ------------------------------------------------
// ...do something with the data;
// launch a dialog for instance and let the user
// modify the target values
// ------------------------------------------------

// alloc structures; write e.g. 5 new target values
psTv = new sNcGchkTargetValue[5];
for (int i = 0; i < 5; i++)
{

     // set values
     psTv[i].fRefValue = 1.1f;
     psTv[i].fLowTol   = 0.1f;

     // ...
} // eofor

SAFEARRAYBOUND rgb [] = {
     (5 * sizeof(sNcGchkTargetValue)), 0
};

// create a 1-dimensional safearray
SAFEARRAY* psa = ::SafeArrayCreate(VT_UI1, 1, rgb);
if (psa)
{
     void* pData;
     ::SafeArrayAccessData(psa, (void**)&pData);

     // serialize the data into the char array
     memcpy(pData, psTv, rgb->cElements * sizeof(char));

     // release
     ::SafeArrayUnaccessData(psa);

     // now pack the data into the VARIANT
     VARIANT arg;
     VariantInit(&arg);
     arg.vt     = VT_UI1 | VT_ARRAY;
     arg.parray = psa;

     // transmit
```

```
                if (FALSE == pICf->SetTargetValues(arg))
                    TRACE0("error!\n");
            } // eoif (psa)

            delete [] psTv;
            break;
        } // eof case NC_FCT_GCHK

    default:
        break;
    }

} // eof function ModifyTargetValues()
```

7.9.8 Check Function Result

The following source code sample demonstrates how to read out the result of a check function after execution in automatic mode. Function Identify Bar Code uses a simple structure to transfer its result data, function Evaluate Classes an array of result values.

```
void ReadResult(ICheckFunction* pICf)
{

    // read current result data
    VARIANT arg = pICf->GetCurrentResult();

    // error check
    // data is transfered between server and controller hidden
    // in a safearray
    if (arg.vt != (VT_UI1 | VT_ARRAY))
    {
        TRACE0("error: invalid return value!\n");
        return;
    } // eoif

    switch(pICf->GetFunctionId())
    {
    case NC_FCT_BCI: // Identify Bar Code, define in ncauto.h
        {
            sNcBciResult* psRes;       // declared in ncauto.h

            // access data
            ::SafeArrayAccessData(arg.parray, (void**)&psRes);
            TRACE1("barcode = %s\n", psRes->szString);

            // release access
            ::SafeArrayUnaccessData(arg.parray);

            // notify system that safearray is no longer needed
            ::SafeArrayDestroy(arg.parray);

            break;
        } // eof case NC_FCT_BCI

    case NC_FCT_CLE: // Evaluate Classes, define in ncauto.h
        {
            SAFEARRAY* psa = arg.parray;
            unsigned int uiNumOfElements =
                (psa->rgsabound[0].cElements / sizeof(sNcCleResult));

            TRACE1("%u struct elements\n", uiNumOfElements);
            if (0 == uiNumOfElements)
                return;
                sNcCleResult* psRes; // declared in ncauto.h

                // access data
                ::SafeArrayAccessData(arg.parray, (void**)&psRes);

                for (unsigned int ui = 0; ui < uiNumOfElements; ui++)
                {
                    // output: class string and quality
                    TRACE2("class %s = %f\n", psRes[ui].szName,
                            psRes[ui].fQuality);
                } // eofor

                // release access
                ::SafeArrayUnaccessData(arg.parray);

                // notify system that safearray is no longer needed
                ::SafeArrayDestroy(arg.parray);
                break;
            } // eof case NC_FCT_CLE

        default:
            break;

        } // eof switch
    } // eof function ReadResult()
```

7.9.9 Read Image

The following source code sample demonstrates how to read a gray level image from the image stack of an individual check into the controller.

```
void ReadImageFromStack(ISingleCheck* pISc)
{

    // first determine the number of available images
    int iNumOfImages = pISc->GetNumOfImages();
    if (0 == iNumOfImages)
        // nothing todo
        return;

    // we have at least one image on the runtime data stack,
    // read image at index zero
    // read image properties
    VARIANT argProp = pISc->GetImageProp(0);

    // error check
    if (argProp.vt != (VT_UI1 | VT_ARRAY))
    {
        TRACE0("error: invalid return data!\n");
        return;
    } // eoif

    sNcImageProp* psProp; // declared in ncauto.h

    // access data
    ::SafeArrayAccessData(argProp.parray, (void**)&psProp);

    // verify image properties
    if ((psProp->uiXSize > 0) && (psProp->uiYSize > 0))
    {

        // continue evaluation with verification of
        // image data
        VARIANT argData = pISc->GetImageData(0);
        if (argData.vt != (VT_UI1 | VT_ARRAY))
```

```
            {
                TRACE0("error: invalid return data!\n");
                ::SafeArrayUnaccessData(argProp.parray);
                ::SafeArrayDestroy(argProp.parray);
                return;
            } // eoif

            // transfer grayvalue pixel array
            char* pbyData;
            ::SafeArrayAccessData(argData.parray, (void**)&pbyData);

            // now work with the image data...
            // ...
            // e.g. create a DIB and display it

            // release access
            ::SafeArrayUnaccessData(argData.parray);
            ::SafeArrayUnaccessData(argProp.parray);

            // notify system that safearray is no longer needed
            ::SafeArrayDestroy(argProp.parray);
            ::SafeArrayDestroy(argData.parray);
        } // eoif

    } // eof function ReadImageFromStack()
```

8 Check Routine in XML Format

This chapter describes the format created when check routines are stored as XML files, and possible applications.

8.1 Introduction

As of NeuroCheck version 5.1, it is possible to store (i.e. to export) check routines in XML format. This operation is described in the **User Manual** and the NeuroCheck online help. XML check routine files (extension: "`*.chrx`") contain the same information as check routine files in the binary format (`*.chr`) which is used as the default so far. In contrast to the binary format, XML format is readable for both humans and for processing programs. Chapter 8.2 gives a detailed description of the XML format of the NeuroCheck check routine data.

8.1.1 What is XML ?

XML stands for e**X**tensible **M**arkup **L**anguage; it is a meta language for the definition of document types. The language was standardized by the World Wide Web Consortium (W3C at `www.w3c.org`). XML provides a strict separation between document structure, contents and visual rendering of the contents. The advantage of XML is that it is easily understandable for both people and processing programs (parsers). Thus, XML is ideally suited for exchanging data between different platforms and for medium term or long term storage of important documents. Already, this technology has established itself as a standard in the IT world for these purposes.

8.1.2 Possible applications of XML format

Using XML as the check routine file format offers a number of new possibilities for NeuroCheck users. The main application areas are:

- **Documentation:**
  An XML file can be viewed in a standard browser such as Microsoft Internet Explorer. Together with additional formatting information (provided by an XSL file), the view of the check routine thus created can be printed and archived for documentation purposes. Depending on the content of the XSL file, the check routine data can be viewed at various levels of detail. For more information on XSL files please refer to chapter 8.3.
- **Change monitoring:**
  Because of the clear structure of the XML file, similar check routines can be compared visually using an editor. This was not possible in the binary format (`*.chr`) used so far. This way, you can quickly determine where a target value or parameter in a check function was changed.

8.2 Description of the XML format of check routines

This chapter describes the XML format of the NeuroCheck check routine data. The focus is on the fundamental structure of the format and on hints to particular features. The necessary formal definitions (e.g. the IDs and value ranges of the check function parameters) can be found in various files in the subdirectory `\Docs\Xml\` of the NeuroCheck installation directory. For a deeper understanding of the format some knowledge of XML is required.

8.2.1 General description of the structure of XML files

The check routine XML format is based on the definition of XML 1.0 according to the recommendation of the World Wide Web Consortium (W3C) as of October 6, 2000. The formal specification can be found on the website `http://www.w3.org/TR/REC-xml`. Please note that the definition is under constant review by the committee. Since this specification is quite formal, the most important terms of this meta language will be briefly described in the following.

- An XML file usually consists of a prolog and a main part.
- The prolog gives general information concerning the XML file. These are e.g. the XML declaration, references to external layout information or document type definitions. They can be recognized by the leading character sequence `<?` or `<!`. Example:
  ```
  <?xml version="1.0"?>
  ```
- The main part of an XML file contains the data structured in elements.
- Each element is enclosed in a start tag and an end tag. The area between the tags is the element content. The content can consist of an arbitrary number of arbitrary characters (excluding > and < and some other reserved characters). Usually the content is text, i.e. words and numbers. Some examples of elements:
  ```
  <element>data</element>

  <element2>1.95583</element2>

  <address>552 Northern Avenue, IL 60142-5298</address>
  ```
- Elements can contain attributes that are assigned inside the start tag. The value assigned to the attribute must be enclosed in quotation marks. Example:
  ```
  <element id="100" type="int">42</element>
  ```
- Summary of the nomenclature using an example:
  ```
  <x y="z">abc</x>
  ```
  With:

x:	element name
abc:	element content
y:	attribute name
z:	attribute value

- Elements can be nested. This means that an element can contain one or more other elements. The nesting is always hierarchic as with a tree structure. Elements must not overlap. Example:
  ```
  <main_element>
  ```

```
        <e2>data</e2>
        <e2>other data</e2>
    </main_element>
```

The elements enclosed in another element are called child elements of the enclosing element.

- Comments, i.e. parts of the XML file that are only intended to aid in the interpretation of the file and do not contain any data, are surrounded by the marks `<!--` and `-->`. Example:

  ```
  <!-- Beginning of public data -->
  ```

- Data within the XML file that cannot be displayed – usually binary data – are within so-called CDATA sections that begin with `<![CDATA[` and end with `]]>`.

8.2.2 Overview of the structure of the XML file of a check routine

NeuroCheck check routines in XML format are stored in files with the extension `*.chrx`. These files contain the same information as the corresponding check routine files in binary format (`*.chr`). The format of the XML check routine file adheres to the syntactic specification of the XML meta language. The formal definition of the used elements can be found in the files in the subdirectory `\Docs\Xml\` of the NeuroCheck installation directory. With some basic knowledge of XML and experience with NeuroCheck, the arrangement of the XML elements is easily understandable allowing for a visual interpretation of the XML file or a comparison of two similar XML check routines in an editor.

The prolog part of the XML file contains a reference to an XSL file which is necessary for the visual rendering of the data. Please refer to chapter 8.3.1 for more information. The reference may look as follows:

```
<?xml-stylesheet href="C:\Ncheck\XSL\CHRX_Detailed.xsl" type="text/xsl"?>
```

The main element of the XML file contains the check routine data, represented by an XML element by the name of `neurocheck_check_routine`. This and its constituent parts will be explained in the following sections.

8.2.3 Sample

First we will show you a sample check routine in XML format. For clarity's sake, the sample has been reduced to the essentials. Some elements were skipped and element attributes were left out. The entire check routine can be found as file `Nc_Sample.chrx` in the subdirectory `\Docs\Xml\` of the NeuroCheck installation directory.

```
<neurocheck_check_routine xmlns="http://www.neurocheck.com/xml">
  <!-- ############  Header information. Do not change !  ############# -->
  <header name="Header information">
     <meaning_of_contents>NeuroCheck Check Routine</meaning_of_contents>
     <created_by_name>NCheck51.exe</created_by_name>
     <created_by_dir>C:\Program Files\Ncheck51\</created_by_dir>
     <created_by_version>5, 1, 1038 [4]</created_by_version>
     <file_version name="Version identification">0x5111</file_version>
     <copyright>NeuroCheck GmbH, D-71686 Remseck, Germany</copyright>
     ...
  </header>
  <!-- #####################  Check routine data  ##################### -->
  <body name="Check routine data">
     <cr_properties name="Check routine properties">
        <cr_description name="Description of check routine">
           <cr_name>NeuroCheck XML sample</cr_name>
           <cr_comment>Sample check routine</cr_comment>
           <cr_description_text>
              <binary_stream_encapsulation>
                 <stream>
                    <span xmlns="">This check routine contains two SC.
                       <ul>
                          <li>The first single check contains ...</li>
                          <li>The second single check contains ...</li>
                       </ul>
                    </span>
                 </stream>
                 <plain>
                    <![CDATA[This check routine contains two SC.<ul><li>The
                             first single check contains ...</li><li>The se
                             cond single check contains ...</li></ul> ]]>
                 </plain>
              </binary_stream_encapsulation>
           </cr_description_text>
           ...
        </cr_description>
        <cr_communication name="Communication">
           <cr_communication_id>0</cr_communication_id>
        </cr_communication>
        ...
     </cr_properties>
     <sc_list name="List of single checks">
        <!-- ================  Bar code identification  =============== -->
        <single_check name="Single check" type="0" index="0">
           <sc_properties name="Single check properties">
              <sc_description name="Description of single check">
                 ...
              </sc_description>
              ...
           </sc_properties>
           <cf_list name="Check function list">
              <!-- ++++++++++++++++  Transfer Image  ++++++++++++++++ -->
              <check_function id="508" display="Transfer Image">
                 <cf_properties name="Check function properties">
                    <check_function_description>
                       ...
                    </check_function_description>
                 </cf_properties>
                 <cf_parameter name="List of parameters">
```

```
                    <parameter id="50801" name="Image source"
                                     display="Bitmap file">1</parameter>
                    <parameter id="50804" name="Bitmap file name">
                       C:\Program Files\Ncheck\Examples\Demo.bmp
                    </parameter>
                    ...
                 </cf_parameter>
              </check_function>
              <!-- +++++++++++++++++++  Define ROIs  +++++++++++++++++ -->
              <check_function id="512" display="Define ROIs">
                 ...
                 <cf_parameter name="List of parameters">
                    <parameter id="51201" name="List of ROIs" type="list">
                       <parameter id="51202" name="Region" type="vector">
                         <parameter id="51204" name="Group">0</parameter>
                         <parameter id="51208" name="UpL X">0</parameter>
                         <parameter id="51209" name="UpL Y">55</parameter>
                         <parameter id="51210" name="Width">200</parameter>
                         <parameter id="51211" name="Height">29</parameter>
                       </parameter>
                    </parameter>
                 </cf_parameter>
              </check_function>
              <!-- ++++++++++++++++  Identify Bar Code  +++++++++++++++ -->
              <check_function id="536" display="Identify Bar Code">
                 ...
                 <cf_parameter name="List of parameters">
                    <parameter id="53601" name="Type" display="EAN 8">2
                                                               </parameter>
                    <parameter id="53602" name="L. distance">5</parameter>
                    <parameter id="53603" name="Smoothing">5</parameter>
                    ...
                 </cf_parameter>
                 <cf_target_values name="List of target values">
                    <parameter id="53607" name="Compare with target value"
                                            display="Yes">1</parameter>
                    <parameter id="53608" name="Target">218641</parameter>
                 </cf_target_values>
              </check_function>
           </cf_list>
        </single_check>
        <single_check>
           ...
        </single_check>
     </sc_list>
     <cr_automatic_screen_config>
        ...
     </cr_automatic_screen_config>
  </body>
</neurocheck_check_routine>
```

8.2.4 Element <neurocheck_check_routine>

The element `neurocheck_check_routine` contains all data of the NeuroCheck check routine and information about the NeuroCheck installation. Its `xmlns` attribute indicates that this element and all its child elements are defined in the NeuroCheck name space.
The element is divided into the child elements `header` and `body`.

8.2.5 Element <header>

The header element contains important information about the NeuroCheck installation. Its child element file_version, for example, is used to manage future changes of the format of the XML file. Do not edit the structure or the contents of the child elements of the header element.

8.2.6 Element <body>

The body element contains all data of the NeuroCheck check routine. Its child elements are divided into logical blocks, e.g. into the elements cr_properties, sc_list and cr_automatic_screen_config etc. with their respective child elements.
The element cr_properties contains all the properties of the check routine.
The individual checks are listed as child elements single_check of sc_list.

8.2.7 Element <single_check>

The single_check element represents individual checks as well as start and end actions. Each element is labeled with the attributes name and type. The child elements of a check are divided into logical blocks, e.g. into the elements sc_properties and cf_list etc. with their respective child elements. The check functions of the individual checks can be found in the child elements check_function of cf_list.

8.2.8 Element <check_function>

The check_function element represents a NeuroCheck check function. The type of the check function can be recognized by its attributes category, id and display (see also the corresponding files in the subdirectory \Docs\Xml\ of the NeuroCheck installation directory). All data of the check function is represented by the child elements of this element, e.g. by the element cf_properties with its respective child elements. The parameter settings and target values of the check function can be found in the child elements cf_parameter and cf_target_values.

8.2.9 Elements <cf_parameter> and <cf_target_values>

These two elements contain only child elements of the type parameter. All parameters and target values that can be configured in the Parameter and Target Value dialogs of the respective check function are represented here.

8.2.10 Element <parameter>

The parameter element represents the settings of a parameter or target value in a check function. The parameter can be identified by its attribute id (see also the corresponding files in the subdirectory \Docs\Xml\ of the NeuroCheck installation directory).
The same applies to the elements input_parameter, output_parameter and window_parameter, which represent the parameter settings of some other NeuroCheck objects.

Parameter elements where the attribute `type` has the value `type="vector"`, `type="list"` or `type="array"` contain arrays with a fixed or variable number of child elements. This is the case, for example, for lists of objects or group-wise parameter settings.

8.2.11 Notes on element attributes

In the following, some attributes are described that have a similar meaning within several elements.
Example 1:

```
<cf_write_to_file name="File output" display="Yes">1</cf_write_to_file>
```

Example 2:

```
<parameter id="53601" name="Barcode type" display="EAN 8">2</parameter>
```

Example 3:

```
<parameter id="50607" name="Manual Threshold" disabled="1">100</parameter>
```

- **Attribute `name`**
  This attribute contains a plain language description of the element (in example 1 `"File output"`). This description is easier to understand for a human than the element name (`"cf_write_to_file"` in example 1). In the case of check function parameters (example 2), the attribute contains the designation of the parameter to be set in the dialog (`"Barcode type"`). The content of this attribute depends on the language version of NeuroCheck (English/German).
- **Attribute `display`**
  This attribute contains a description of the element‘s content (`"Yes"` in example 1, `"EAN 8"` in example 2). This description is easier to interpret by a human than the element‘s content itself (`"2"` in example 2). The content of this attribute depends on the language version of NeuroCheck (English/German).
- **Attribute `disabled`**
  Elements containing the attribute `disabled="1"` are ignored when the check routine is executed. In the above example, this could be because another parameter, the threshold determination method, is set to automatic computation, and not to manual input.

8.2.12 Element <binary_stream_encapsulation>

`binary_stream_encapsulation` elements contain complex check routine data that might conflict with the XML syntax specification. This affects binary data within PlugIn parameter sections, binary image data or certain HTML tags within the description texts of NeuroCheck objects.
The child elements `data` and `plain` contain the data in a format usable by NeuroCheck, whereas the element `stream` is used to display the data in XSL.

8.2.13 Purpose of the individual parts

The individual parts of the NeuroCheck check routine XML file have different purposes. The element structure and the element contents (data) are of course the most important parts. Some parts of the XML file only serve to enhance clarity and to facilitate visual interpretation of the data when viewing them in an editor. Other parts are helpful for different views in the browser. The following table shows which part is used for what.

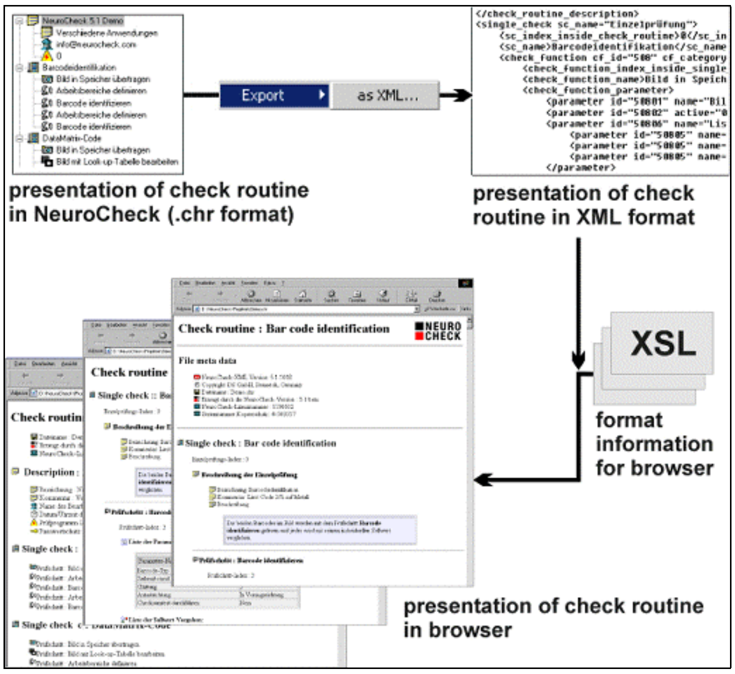
	Visual interpretation of raw XML data in editor	Browser view (with XSL)
Element structure	X	X
Element contents	X	X
Prolog of XML file		X
CDATA sections		
Attribute `id`		
Attribute `index`		
Attribute `name`	X	X
Attribute `display`	X	X
Attribute `disabled`	X	X
Attribute `type`		X
Attribute `category`		X
Comments in XML file	X	
Line breaking and indentation in XML file	X	

For an XML file created externally for import into NeuroCheck this means for example, that many attributes are not required.

8.3 Using XSL files as rendering filter for the XML data

8.3.1 Purpose of the XSL files

XML provides a strict separation between document structure, contents and visual formatting of the contents. Therefore, an XSL file is necessary for the visualization of XML data in the browser. Like XML, XSL is a standard by the W3C; it stands for e**X**tensible **S**tyle sheet **L**anguage. An XSL file acts as a filter that determines how the data of the XML file is to be rendered. Internally, the browser uses the XSL instructions to perform a transformation from XML to a HTML page.


Export
as XML...
presentation of check routine in NeuroCheck (.chr format)
presentation of check routine in XML format
XSL
format information for browser
presentation of check routine in browser

```
</check_routine_description>
<single_check sc_name="Einzelprüfung">
    <sc_index_inside_check_routine>0</sc_in
    <sc_name>Barcodeidentifikation</sc_name
    <check_function cf_id="508" cf_category
        <check_function_index_inside_single
        <check_function_name>Bild in Speich
        <check_function_parameter>
            <parameter id="50801" name="Bil
            <parameter id="50802" active="0
            <parameter id="50806" name="Lis
                <parameter id="50805" name=
                <parameter id="50805" name=
                <parameter id="50805" name=
            </parameter>
```

Figure 19: Utilization of XSL file as rendering filter for XML data

The **User Manual** and the NeuroCheck online help describe how an XSL file can be assigned to a CHRX file when exporting a check routine.
There are alternatives to using XSL files to render XML data, e.g. using a JavaScript built into a HTML page. These methods are not described here.

8.3.2 Creation of your own XSL files

Depending on the content of the XSL file, the check routine data can be viewed at various levels of detail. In addition to the check routine data, the XSL rendering can also contain arbitrary texts in XHTML and graphics.
Every user can create his or her own XSL files. We recommend using an existing XSL file as a template, saving it under a different name and changing it. Included with NeuroCheck, you can find a number of XSL files which provide different renderings of check routines. You can find the files with the extension `".xsl"` in the subdirectory `\XSL\` under the NeuroCheck installation directory. You can choose any directory for storing your own XSL files; however, for clarity's sake we recommend using the same directory.

☞ XSL adheres to the syntax specification of the XML meta language. Please note that most browsers react with an error message to even the smallest mistake in an XSL file's syntax without displaying anything else.

8.3.3 Notes on the included XSL files

In this section important details of the XSL files will be explained. For more information concerning XSL, XSLT, XPath, stylesheets or XHTML please refer to the common literature.

- **Compatibility with Internet Explorer versions**
  To ensure compatibility with the older versions of Microsoft Internet Explorer (IE), the included XSL files stick to an older version of the XSL syntax. The reference to the XSL namespace can be found at the top of each XSL file:
  ```
  <xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  ```
  This XSL syntax is recognized by IE versions 5.0 and higher. The recent version of the XSL syntax is only recognized by IE 6.0 and higher. If you want to make use of some of the latest features of XSL, you have to change the namespace declaration as follows:
  ```
  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  ```
  Please note that for this namespace the syntax of some XSL instructions, for example the syntax of the `test` condition, has to be changed.
- **Element `<xsl_description>`**
  This element is situated near the top of the XSL file within the root node template. This element and its child elements are evaluated when the XSL file is incorporated into NeuroCheck to display its contents in the list of XSL files. These elements should therefore be used in their existing form and filled with your own data when you create your own XSL files.
- **`<xsl:template>` elements**
  Display of the check routine data is done using XSL templates. From a programmer's point of view, these can be seen as subroutines that call each other. It is advantageous to use separate templates for the check routine, an individual check, a check function and the parameters. In the following example you will see how the template for an individual check makes use of the templates of all subordinated check functions:
  ```
  <xsl:template match="nc:single_check">
      ...
      <xsl:apply-templates select="nc:cf_list/nc:check_function"/>
  </xsl:template>
  ```
- **Access to XML check routine data**
  In order to access the check routine data in the XML file you can use the syntax of the XPath language, which again is too complex to be explained in detail here. The content of an XML element is evaluated by the XSL element `<xsl:value-of/>` and can thus be displayed in the browser. The assignment of the attribute `select` tells XSL which XML element or attribute should be evaluated.
  ```
  <xsl:template match="nc:single_check">
      <xsl:value-of select="nc:sc_properties/nc:sc_descript/nc:sc_name"/>
      ...
  </xsl:template>
  ```
  The access to an element is similar to browsing a directory tree by using relative XPath specifications starting from the current XML element. In this example, the template processing the XML element `single_check` first accesses its child element

sc_properties, then that one's child element sc_descript and then that one's child element sc_name. Please note that you have to use the "nc:" prefix because all elements are defined in the NeuroCheck namespace.

It is not always only the content of an element that is evaluated. For example, the attributes name and display often allow for a better rendering of the element and its values. For example, if the following XML element

```
<parameter name="Barcode type" display="EAN 8">2</parameter>
```

is evaluated with the following XSL code

```
<xsl:template match="nc:parameter">
    The parameter
    <xsl:value-of select="@name"/>
    has been set to
    <xsl:value-of/>
    which means
    <xsl:value-of select="@display"/>
</xsl:template>
```

the result will be:

```
The parameter Barcode type has been set to 2 which means EAN 8
```

- **Branchings**
  Using the XSL elements <xsl:if>, <xsl:choose>, <xsl:when> or <xsl:otherwise>, it is possible to make XSL statements conditional, e.g. dependent on the value of XML attributes. Example:

```
<xsl:template match="nc:parameter">
    <xsl:choose>
        <xsl:when test=".[@type='list']">
            <!-- do something with this list -->
            ...
        </xsl:when>
        <xsl:when test=".[@type='vector']">
            <!-- do something with this vector -->
            ...
        </xsl:when>
        <xsl:otherwise>
            <!-- this seems to be neither list nor vector -->
            ...
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

- **Including images**
  It is recommended to store included image files in the subdirectory \XSL\Images\ under the NeuroCheck installation directory and access them using an absolute directory specification. For this, use the XML element created_by_dir which contains the NeuroCheck installation directory, e.g. C:\Program Files\Ncheck\. Example:

```
<img>
   <xsl:attribute name="src">
      <xsl:value-of select="//nc:created_by_dir"/>\XSL\Images\clock.gif
   </xsl:attribute>
</img>
```

The absolute directory and the file name are concatenated by XSL.

- **HTML**
  If you want to use HTML tags in the XSL file, you should use XHTML instead. It is very similar to HTML, but has a stricter syntax. This means, for example, that all start tags have to be closed properly with end tags. For example:

wrong	correct
<img src="my_image.gif">	<img src="my_image.gif"**/**>
<H2>Title</H2>	<**h2**>Title</**h2**>
<br>	<br**/**>

<ul>
   <li>One
   <li>Two
</ul>
	<ul>
	   <li>One**</li>**
	   <li>Two**</li>**
	</ul>

- **Cascading Style Sheets (CSS)**
  It is possible to make use of Cascading Style Sheets from within the XSL file. Insert references to such files in the `<html><head>` section of the root node template. Example:

```
<link rel="Stylesheet" type="text/css" media="screen">
   <xsl:attribute name="href">
      <xsl:value-of select="//nc:created_by_dir"/>\XSL\nc_screen.css
   </xsl:attribute>
</link>
```

9 Quick Reference

The following pages summarize all functions and data structures relevant for programming user extensions to NeuroCheck with their parameters. For details on how to use the functions and interpret their results please refer to the previous chapters.

9.1 Plug-In Interface Administrative Functions and Structures

DLLInfo()

```
extern "C" void WINAPI DllInfo(HWND hwndMain);
```

Displays about box of plug-in DLL. Required to be explicitly exported from plug-in DLL.

Parameters	
`hwndMain`	Handle of the NeuroCheck application window.

PI_GetHelpFilePath()

```
extern "C" BOOL WINAPI PI_GetHelpFilePath(LPSTR lpszPath)
```

Informs NeuroCheck about existence and name of a help file for the plug-in DLL. Required to be explicitly exported from plug-in DLL.

Parameters	
`lpszPath`	Address of a buffer with size `_MAX_PATH` for returning the full path name of the help file for the plug-in DLL.

Return Value	
`TRUE`	A help file has been provided for the DLL and its name returned in `lpszPath`.
`FALSE`	No help file has been provided.

PI_MenuCommand()

```
extern "C" void WINAPI PI_MenuCommand(HWND hwndMain,
                      unsigned int uiCmdId)
```

Handles calls to plug-in menu items in the **Tools** menu. Required to be explicitly exported from plug-in DLL.

Parameters	
`hwndMain`	Handle of the NeuroCheck application window.
`uiCmdID`	ID of selected menu item (range `0xD000 - 0xD099`)

PI_GetNumberOfFcts()

```
extern "C" unsigned int WINAPI PI_GetNumberOfFcts(void)
```

Informs NeuroCheck about the number of check functions in the plug-in DLL. Required to be explicitly exported from plug-in DLL.

Return Value

Returns the number of check functions for which NeuroCheck has to allocated function info blocks.

PI_GetFctCaps()

```
extern "C" BOOL WINAPI PI_GetFctCaps(
                      unsigned short int uiIndex,
                      sPI_FCT_DESC* const psFctDesc)
```

Informs NeuroCheck about the properties of a check function from the plug-in DLL. Required to be explicitly exported from plug-in DLL.

Parameters

`uiIndex`	Index of plug-in check function (range `0` to return value of function `PI_GetNumberOfFcts()` minus `1`)
`psFctDesc`	Address of function info block. The function has to fill this data structure with information about the function designated by `uiIndex`

Return Value

`TRUE`	Function exists and its properties have been stored in `*psFctDesc`.
`FALSE`	Value of `uiIndex` has been outside of the valid range.

sPI_FCT_DESC

```
typedef struct
{
    // version control
    unsigned int                uiStructSize;

    // data
    unsigned int                uiFunctionId;
    unsigned int                uiHelpContext;
    unsigned int                uiNumOfInputData;
    unsigned int                uiNumOfOutputData;

    unsigned int                uiSizeOfParameter;
    unsigned int                uiNumOfViews;
    unsigned int                uiDataOutput;

    // pointer
    // - data
    int*                        piTypeOfInputData;
    int*                        piTypeOfOutputData;
    int*                        piMaskFileOutput;
    int*                        piMaskRs232Output;

    // - strings
    LPSTR                       lpszFunctionName;
    LPSTR                       lpszDescription;
```

```
        LPSTR                           lpszHintPos;
        LPSTR                           lpszHintNeg;
        LPSTR*                          alpszCustomViewNames;
        LPSTR*                          alpszInputViewNames;
        LPSTR*                          alpszOutputViewNames;

        // - functions
        PFNPlugInFctInitialize          pfnInitialize;
        PFNPlugInFctExecute             pfnExecute;
        PFNPlugInFctUnInitialize        pfnUnInitialize;
        PFNPlugInFctParameterDlg        pfnParameterDlg;
        PFNPlugInFctViewCreate          pfnViewCreate;
        PFNPlugInFctViewDestroy         pfnViewDestroy;
    }
        sPI_FCT_DESC;
```

Describes capabilities and properties of a plug-in check function.

PI_GetNumberOfDataTypes()

```
extern "C" unsigned int WINAPI PI_GetNumberOfDataTypes(void)
```

Informs NeuroCheck about the number of data types in the plug-in DLL.

Return Value

Returns the number of data types for which NeuroCheck has to allocated info blocks.

PI_GetDataTypeDesc()

```
extern "C" BOOL WINAPI PI_GetDataTypeDesc(
                      unsigned short int uiIndex,
                      sPI_TYPE_DESC* const psTypeDesc)
```

Informs NeuroCheck about the properties of a custom data types from the plug-in DLL.

Parameters

`uiIndex`	Index of plug-in data type (range `0` to return value of function `PI_GetNumberOfDataTypes()` minus `1`)
`psTypeDesc`	Address of info block. The function has to fill this data structure with information about the data type designated by `uiIndex`

Return Value

`TRUE`	Data type exists and its properties have been stored in `*psTypeDesc`.
`FALSE`	Value of `uiIndex` has been outside of the valid range.

sPI_TYPE_DESC

```
typedef struct
{
    // version control
    unsigned int                uiStructSize;

    // data
    unsigned int                uiTypeId;
    int                         iIconIndex;
```

```
        // - strings
        LPSTR                          lpszTypeDesc;

        // - function pointer
        PFNPlugInTypeDataOutput        pfnDataOutput;
        PFNPlugInTypeDestroy           pfnTypeDestroy;
    }
        sPI_TYPE_DESC;
```

Describes capabilities and properties of a plug-in data type.

9.2 Plug-In Check Function Declaration

This section lists the declarations required for the routines making up a plug-in check function. The routines of your plug-in check functions must match the declarations exactly except for the name.

Initialization Routine

```
BOOL WINAPI InitFunction(
                          sPI_CONTEXT_INFO* const psContext,
                          void* const pParameter);
```

Performs initialization necessary for a plug-in check function when it is first inserted into a check routine (or loaded together with a check routine).

Parameters

psContext	Pointer to structure containing context information for function.
pParameter	Address of parameter block reserved for the function.

Return Value

TRUE	Initialization has been carried out successfully.
FALSE	An error occurred during initialization.

Deinitialization Routine

```
void WINAPI DeInitFunction(
             sPI_CONTEXT_INFO* const psContext,
             void* const pParameter);
```

Peforms necessary cleaning up for a plug-in check function when it is removed from the check routine (or when the check routine is unloaded).

Parameters

psContext	Pointer to structure containing context information for function.
pParameter	Address of parameter block reserved for the function.

Execution Routine

```
BOOL WINAPI ExecuteFunction(
             sPI_CONTEXT_INFO* const psContext,
             void* const pParameter,
                 void* const * const ppvDynInput,
                 void*       * const ppvDynOutput);
```

Performs the actual computation; called whenever NeuroCheck encounters the plug-in check function object during execution of a check routine.

Parameters

`psContext`	Pointer to structure containing context information for function.
`pParameter`	Address of parameter area reserved for the function.
`ppvDynInput`	Address of array with `PI_MAXDYNDATA` pointers to input data objects. Only as many pointers are valid as have been requested in the function info block.
`ppvDynOutput`	Address of array with `PI_MAXDYNDATA` pointers to output data objects. Only as many pointers can be used as have been requested in the function info block.

Return Value

`TRUE`	Execution has been carried out successfully.
`FALSE`	An error occurred during execution. The current check will terminate with status `not O.K.`

Parameter Dialog Routine

```
BOOL WINAPI ParameterDialog(
             HWND hwndApp,
             sPI_CONTEXT_INFO* const psContext,
             void* const pParameter,
             void* const * const ppvInputData);
```

Handles the parameter input dialog of the check function. This routine is not required to exist for a plug-in check function.

Parameters

`hwndApp`	Handle for NeuroCheck application window.
`psContext`	Pointer to structure containing context information for function.
`pParameter`	Address of parameter block reserved for the function.
`ppvInputData`	Address of array with `PI_MAXDYNDATA` pointers to input data objects. Only as many pointers are valid as have been requested in the function info block.

Return Value

`IDOK`	Dialog has been left with OK, parameter data has changed.
`IDCANCEL`	Dialog has been left with Cancel, no change of parameter data.

Custom Visualization Routines

Handle the custom visualization of the check function. These routines are not required to exist for a plug-in check function.

```
BOOL WINAPI ViewCreate(
            sPI_CONTEXT_INFO* const psContext,
            sPI_VIEW_INFO* const psViewInfo,
            HANDLE* hView,
            void* const pParameter,
            void* const * const ppvInputData,
            void* const * const ppvOutputData);
```

Parameters

psContext	Pointer to structure containing context information for function.
psViewInfo	Pointer to structure containing view information.
hView	Address to return the bitmap handle for .
pParameter	Address of parameter block reserved for the function.
ppvInputData	Address of array with PI_MAXDYNDATA pointers to input data objects. Only as many pointers are valid as have been requested in the function info block.
ppvOutputData	Address of array with PI_MAXDYNDATA pointers to output data objects. Only as many pointers are valid as have been requested in the function info block. If the execution of the function returned FALSE, the output data will not be valid.

Return Value

TRUE	Bitmap for view created successfully and returnd in hView.
FALSE	Creation of view failed, hView will be ignored.

```
BOOL WINAPI ViewDestroy(
            sPI_VIEW_INFO* const psViewInfo,
            HANDLE* hView);
```

Called to release the view data allocated by `ViewCreate`.

Parameters

psViewInfo	Pointer to structure containing view information.
hView	Handle to bitmap allocated in ViewCreate.

Return Value

TRUE	Bitmap passed in hView released successfully.
FALSE	Function failed.

9.3 Plug-In Data Type Declaration

This section lists the declarations required for the routines making up a plug-in data type. The routines of your plug-in check functions must match the declarations exactly except for the name.

Data Output Routine

```
BOOL WINAPI DataOutput(sPI_TARGET_INFO* const psTargetInfo,
                       void* const pbyData,
                       sPI_DATAOUTPUT_INFO* psDataContainer);
```

Handles data output of a custom data object in automatic mode if output is activated for the plug-in check function using the plug-in data type.

Parameters

`psTargetInfo`	Pointer to structure containing information about the output target (file, serial interface)
`pbyData`	Address of custom data object allocated previously inside the execution routine.
`psDataContainer`	Address of data container used for returning a data sequence NeuroCheck will write to the given output target.

Return Value

`TRUE`	Output data passed successfully in `psDataContainer`.
`FALSE`	No output data available.

Destroy Routine

```
void WINAPI TypeDestroy(void* pbyData);
```

Peforms necessary cleaning up of a custom data object of the plug-in data type when it is removed from the data pool of the individual check.

Parameters

`pbyData`	Address of custom data object allocated previously inside the execution routine.

9.4 Plug-In Data Exchange Structures

sPI_IMAGE

```
typedef struct
{
    unsigned short int    uiWidth;      // width of image
    unsigned short int    uiHeight;     // height of image
    BOOL                  bColor;       // color image?
    BYTE *            pbyGrayValue;     // gray level values
    BYTE *            pbyRedValue;      // values of red channel
    BYTE *            pbyGreenValue;    // values of green channel
    BYTE *            pbyBlueValue;     // values of blue channel
    int               iSource;          // type of image source
    LPCTSTR           lpszSourceName;   // name of image source
}   sPI_IMAGE;
```

Used to exchange gray level image data between NeuroCheck and a plug-in check function.

All elements are read-only. A function may only change values within the image data arrays.

sPI_HISTO

```
typedef struct
{
    unsigned short int    uiThreshold;      //binarization threshold
    unsigned int      auiHistoBuffer[256];  // frequencies of gray levels
}   sPI_HISTO;
```

Used to exchange histogram data between NeuroCheck and a plug-in check function.
All elements may be changed by a plug-in check function.

sPI_LAYER

```
typedef struct
{
    unsigned int      uiCount;          // number of regions in array
    BOOL              bMeasurement[300];// feature validity flags
    LPSTR             lspzMeasurementName[300];// feature names
    unsigned int      uiClasses;        // number of available classes
    LPSTR*            alpszClasses;     // array of class names
    sPI_OBJECT *      pasObject;        // array of pointers to regions
}   sPI_LAYER;
```

Used to exchange region of interest data between NeuroCheck and a plug-in check function.
Except for flags in the `bMeasurement` array all elements should be considered "read-only".

sPI_OBJECT

```
typedef struct
{
    /* ------* Read only *------ */
    int               iType;            // AOI, contour or region
    int               iWidth;           // enclosing rectangle, width
    int               iHeight;          // enclosing rectangle, height
    sPI_CONTOUR*      psContour;        // contour (may be missing)
    sPI_REGION*       psRegion;         // region (may be missing)
    // model geometries
    int               iFitType;         // description of model geometry
    float             fFitParameters[10];// parameters of model geometry

    /* ----------------------------------- */
    /* ------* Modify/Write *------ */
    int               iNumber;          // index of object (for sorting)
    int               iGroupNumber;     // group membership
    BOOL              bValid;           // validity (for screening)
    int               iX;               // enclosing rectangle, left edge
    int               iY;               // enclosing rectangle, top edge
    float             fMeasurement[300];// feature values
    /* -------* Write only *------ */
    int               iShapeType;       // shape type (AOI, polyline)
    int               iSearchReg;       // size of surrounding area
    sPI_POINTS*       psPoints;         // description of new object
}   sPI_OBJECT;
```

Used to describe a single region of interest in an `sPI_LAYER` structure.
For a new object, the „write-only“ and „modify/write“ elements must be specified.

Only „modify/write“ elements can be altered for input objects by a plug-in check function:

`iNumber`	Changes sorting of regions.
`iGroupNumber`	Changes groub membership of regions.
`bValid`	Assigning `FALSE` lets region be removed.
`iX, iY`	Changes position of region.
`fMeasurement[i]`	Changes value of feature `[i]`.

sPI_CONTOUR

```
typedef struct
{
    int      iXStart;     // starting point x coordinate
    int      iYStart;     // starting point y coordinate
    int      iLength;     // number of contour elements
    BYTE *   pbyChain;    // chain code array
}   sPI_CONTOUR;
```

Used to describe the contour of a single region of interest in an `sPI_OBJECT` structure. All elements are to be considered read-only.

sPI_REGION

```
typedef struct
{
    int          iLength;     // length of RLC code
    int*         piX;         // X coordinates of start positions
    int*         piY;         // Y coordinates of start positions
    int*         piXLength;   // lengths of line segment in X direction
}   sPI_REGION;
```

Used to describe the region of a single region of interest in an `sPI_OBJECT` structure. All elements are to be considered read-only.

sPI_POINTS

```
typedef struct
{
    int      iLength;     // array length, i.e. number of points
    int*     piX;         // X coordinates of points
    int*     piY;         // Y coordinates of points
}   sPI_POINTS;
```

Used to describe a newly created object in the `sPI_OBJECT` structure. All elements are to be considered write-only.

sPI_MEASARRAY

```
typedef struct
{
    unsigned int      uiCount; // number of values
    sPI_MEASVALUE *   pasMeasValue; // array with value structures
}   sPI_MEASARRAY;
```

Used to exchange geometrical measurement data between NeuroCheck and a plug-in check function. All elements are to be considered „read-only“.

sPI_MEASVALUE

```
typedef struct
{
    unsigned int uiNumber;    // identification number
    float        fMeasValue;  // actual value
    LPSTR        lpszName;    // description string (may be NULL)
}   sPI_MEASVALUE;
```

Used to describe a single measurement in a `sPI_MEASARRAY` structure. Element `lpszName` should be considered "read-only".

9.5 Plug-In Interface Symbolic Constants

Identification of Data Objects

Constant	Description
`PI_OBJECTAOI`	Region contains enclosing rectangle only.
`PI_OBJECTCONTOUR`	Region also contains contour description.
`PI_OBJECTREGION`	Region also contains contour and region description.

Identification of Model Geometries

Constant	Description
`PI_FIT_CONTOUR`	No specific model geometry calculated.
`PI_FIT_POINT`	Model geometry is a single point.
`PI_FIT_LINE`	Model geometry is a line.
`PI_FIT_CIRCLE`	Model geometry is a circle.

Feature Values

Constant	Description
`PI_ALL_XCOFG`	X coordinate of the center of gravity of the region.
`PI_ALL_YCOFG`	Y coordinate of the center of gravity of the region.
`PI_AOI_XORG`	X coordinate of top left corner of the enclosing rectangle.
`PI_AOI_YORG`	Y coordinate of top left corner of the enclosing rectangle.
`PI_AOI_XDIM`	Width of enclosing rectangle.
`PI_AOI_YDIM`	Height of enclosing rectangle.
`PI_AOI_RATIO`	Ratio of height to width of enclosing rectangle.
`PI_BNDAOI_LAXIS`	Length of principal axis of the region.
`PI_BNDAOI_SAXIS`	Length of secondary axis of the region.
`PI_BNDAOI_LANGLE_180`	Direction of principal axis without regard to orientation (i.e. between 0 and 180°)
`PI_BNDAOI_LANGLE_360`	Direction of principal axis with regard to orientation (i.e. between

	0 and 360°)
`PI_BNDAOI_RADMEAN`	Average radius.
`PI_BNDAOI_RADMIN`	Mimum radius.
`PI_BNDAOI_RADMAX`	Maximum radius.
`PI_BNDAOI_RANGLE`	Angle between minimum and maximum radius.
`PI_ALL_AREA`	Area of region.
`PI_BNDAOI_PERI`	Circumference of region.
`PI_BNDAOI_COMPACT`	Form factor of region (defined as Area/(4 * π * Circumference^2); maximum value 1 for ideal circles, else smaller).
`PI_BNDAOI_FIBRELENGTH`	Approximate length of a line following the shape of the region holding equal distance to both edges.
`PI_BNDAOI_FIBREWIDTH`	Sum of distances from the fibre to both edges.
`PI_BNDAOI_ELONGATION`	Ratio of fibre length to width.
`PI_AOI_ANY`	Region touches any image border.
`PI_AOI_UP`	Region touches top border of image.
`PI_AOI_DOWN`	Region touches bottom border of image.
`PI_AOI_LEFT`	Region touches left border of image.
`PI_AOI_RIGHT`	Region touches right border of image.
`PI_AOI_HOLES`	Number of holes enclosed in the region.
`PI_ALL_MEAN`	Average gray level inside region.
`PI_ALL_MIN`	Minimum gray level inside region.
`PI_ALL_MAX`	Maximum gray level inside region.
`PI_ALL_SIGMA`	Standard deviation of gray levels inside region.
`PI_ALL_CONTRAST`	Maximum difference of gray levels inside region.
`PI_ALL_GRADMEAN`	Average gradient inside region.
`PI_ALL_GRADMAX`	Maximum gradient inside region.
`PI_ALL_GRADSIGMA`	Standard deviation of gray levels inside region.
`PI_GEN_CURV_MEAN`	Average curvature.
`PI_GEN_CURV_SIGMA`	Standard deviation of curvature values.
`PI_GEN_CURV_MIN`	Minimum curvature.
`PI_GEN_CURV_MAX`	Maximum curvature.
`PI_GEN_CURV_CONTRAST`	Maximum amplitude of curvatures along the contour.
`PI_GEN_COR_QUALITY`	Correlation coefficient of region with its template.
`PI_GEN_COR_SUBPIX_X`	X coordinate of region found by subpixel template matching.
`PI_GEN_COR_SUBPIX_Y`	Y coordinate of region found by subpixel template matching.
`PI_GEN_COR_ANGLE`	Angle of rotated template.
`PI_GEN_CLASS`	Number of class determined by classifier.
`PI_GEN_CLASSQUALITY`	

	Quality of classification result (equals the classification certainty).
`PI_POS_OFFSET_X`	Offset X determined by Determine Position.
`PI_POS_OFFSET_Y`	Offset Y determined by Determine Position.
`PI_POS_ROT_ANGLE`	Rotation angle determined by Determine Position.
`PI_POS_PIVOT_X`	Pivot X determined by Determine Position.
`PI_POS_PIVOT_Y`	Pivot Y determined by Determine Position.

9.6 Custom Communication Interface

DLLInit()

```
extern "C" BOOL DllInit (void)
```

Performs necessary initialization for the driver. Required to be explicitly exported.

Return Values

`1`	Successful initialization.
`0`	Error in initialization.

SetupDevice()

```
extern "C" BOOL SetupDevice (HWND hwndAppMain)
```

Can be used to display a setup dialog for the device. The function is not required. If the function exists, it has to be exported explicitly.

Parameters

`hwndAppMain`	Handle for NeuroCheck application window.

Return Values

`1`	Device setup has been changed, `DllInit()` will be called.
`0`	Device setup unchanged.

TestDevice()

```
extern "C" BOOL TestDevice (HWND hwndAppMain)
```

Can be used to display a test dialog for the device. The function is not required. If the function exists, it has to be exported explicitly.

Parameters

`hwndAppMain`	Handle for NeuroCheck application window.

Return Values

`1`	Device is working properly.
`0`	Device is not working.

TestStart()

```
extern "C" BOOL TestStart (void)
```

Checks whether start signal has been received. Required to be explicitly exported.

Return Values

`1`	Start signal has been received.
`0`	No start signal has been received.

GetTypeId()

```
extern "C" BOOL GetTypeId (unsigned short int * pTypeId)
```

Checks whether check routine selection signal has been received and returns ID. Required to be explicitly exported.

Parameters

`pTypeId`	Used to return received check routine ID (`0 - 99999`)

Return Values

`1`	Selection signal has been received and returned in `*pTypeId`.
`0`	No selection signal has been received.

SetCheckResult()

```
extern "C" void SetCheckResult (BOOL bSuccess)
```

Transmits (or buffers) final check result. Required to be explicitly exported.

Parameters

`bSuccess`	Indicates final status of check routine (`FALSE` for `not O.K.`)

TransferFloat()

```
extern "C" void TransferFloat (float fValue)
```

Transmits (or buffers) a floating point value. Required to be explicitly exported.

Parameters

`fValue`	Floating point value to be transmitted.

TransferInt()

```
extern "C" void TransferInt (int iValue)
```

Transmits (or buffers) an integer value. Required to be explicitly exported.

Parameters

`iValue`	Integer value to be transmitted.

TransferString()

```
extern "C" void TransferString (char * pChar, unsigned int uiNumOfChars)
```

Transmits (or buffers) a string. Required to be explicitly exported.

Parameters	
`pChar`	Address of string to be transmitted.
`uiNomOfChars`	Number of valid characters.

Flush()

```
extern "C" BOOL Flush (void)
```

Actuates transmission of buffered values at end of check routine run. Required to be explicitly exported.

Result Values	
`1`	Successful transmission.
`0`	Error in transmission.

9.7 OLE Automation Interface

This section lists the available properties and methods of the OLE automation objects exposed by NeuroCheck. Access to the properties is given as `ro` for "read only" and `rw` for "read/write".

NCApplication Object

Property	Type, Access, Description
`ActiveCamera`	VT_I2, `rw`; reads/sets camera for live view.
`ActiveCameraName`	VT_BSTR, `ro`; returns camera designation for live view.
`ActiveCameraZoom`	VT_I2, `rw`; reads/sets zoom factor for live view.
`ActiveCheckRoutine`	VT_DISPATCH, `ro`; returns the active check routine object or VT_EMPTY if none.
`Application`	VT_DISPATCH, `ro`; returns the application object.
`Caption`	VT_BSTR, `ro`; returns the title of the application window.
`DeviceCount`	VT_I2, `ro`; returns number of devices configured in NeuroCheck, or -1 on error. Takes one argument: - VT_I2 `DeviceType`: Type of device.
`DeviceName`	VT_BSTR, `ro`; returns name of a device configured in NeuroCheck, or empty string on error. Takes two arguments: - VT_I2 `DeviceType`: Type of device.

	- VT_I2 DeviceIndex: Index of device.
ExeMajorVersion	VT_I2, ro; returns major version number (≥ 4).
ExeMinorVersion	VT_I2, ro; returns minor version number (≥ 0).
FullName	VT_BSTR, ro returns the file specification for the application, including path.
Height	VT_I4, rw; reads/sets height of main application window.
IgnoreCommunication	VT_BOOL, rw; reads/sets ignore communication option of application.
InterfaceVersion	VT_I2, ro; returns version number of automation interface.
LastError	VT_I4, ro; returns error number for most recent error.
Left	VT_I4, rw; reads/sets left edge of main application window.
LicenseLevel	VT_I2, ro; returns license level { Premium, Professional, Runtime, Demo }.
LicenseNumber	VT_I4, ro; returns the license number from security key.
Name	VT_BSTR, ro; returns the name of the application.
OperatingMode	VT_I2, rw; reads/sets operating mode {manual, live, automatic}.
Parent	VT_DISPATCH, ro; returns Null.
Path	VT_BSTR, ro; returns path specification for the application.
ReadFromBitmap	VT_BOOL, rw; reads/sets simulate image capture option of application.
SubDeviceCount	VT_I2, ro; returns number of sub devices configured in NeuroCheck, or -1 on error. Takes three arguments: - VT_I2 DeviceType: Type of parent device. - VT_I2 DeviceIndex: Index of parent device. - VT_I2 SubDeviceType: Type of sub device.
SubDeviceName	VT_BSTR, ro; returns name of a sub device configured in NeuroCheck, or empty string on error. Takes four arguments: - VT_I2 DeviceType: Type of parent device. - VT_I2 DeviceIndex: Index of parent device. - VT_I2 SubDeviceType: Type of sub device. - VT_I2 SubDeviceIndex: Index of sub device.
Top	VT_I4, rw; reads/sets top edge of main application window.
Width	VT_I4, rw; reads/sets width of main application window.

`WindowState`	VT_I2, `rw`; reads/sets visual state of main application window.

Method	**Description**
`Execute`	Starts execution of active check routine; returns success or failure. Return type: VT_BOOL Arguments: None
`Open`	Opens existing check routine; returns success or failure. Return type: VT_BOOL Arguments: - VT_BSTR `FileName`: Name of the check routine
`Quit`	Closes check routine and exits application. Return type: None Arguments: None
`ReadDigitalInput`	Reads current state of a digital input. Returns Boolean value for state of input, or VT_EMPTY on error. Return type: VT_VARIANT Arguments: - VT_I2 `BoardIndex`: Index of digital I/O board. - VT_I2 `InputNumber`: Number of input to be read.
`ReadDigitalInputWord`	Reads current state of all (16) digital inputs of one board. Returns decimal value of binary number encoding the state of 16 inputs, or VT_EMPTY on error. Return type: VT_VARIANT Arguments: - VT_I2 `BoardIndex`: Index of digital I/O board.
`ReadDigitalOutput`	Reads current state of a digital output. Returns Boolean value for state of output, or VT_EMPTY on error. Return type: VT_VARIANT Arguments: - VT_I2 `BoardIndex`: Index of digital I/O board. - VT_I2 `OutputNumber`: Number of output to be read.
`ReadDigitalOutputWord`	Reads current state of all (16) digital outputs of one board. Returns decimal value of binary number encoding the state of the 16 outputs, or VT_EMPTY on error. Return type: VT_VARIANT Arguments: - VT_I2 `BoardIndex`: Index of digital I/O board.

`ReadFieldBusInputBit`

Reads current state of an input bit of a field bus device. Returns Boolean value for state of input bit, or VT_EMPTY on error.
Return type: VT_VARIANT
Arguments:
- VT_I2 `BoardIndex`: Index of field bus board.
- VT_I2 `InputNumber`: Number of input bit to be read.

`ReadFieldBusOutputBit`

Reads current state of an output bit of a field bus device. Returns Boolean value for state of output bit, or VT_EMPTY on error.
Return type: VT_VARIANT
Arguments:
- VT_I2 `BoardIndex`: Index of field bus board.
- VT_I2 `OutputNumber`: Number of output bit to be read.

`SetDigitalOutput` Sets state of a digital output. Returns success or failure of set operation.
Return type: VT_BOOL
Arguments:
- VT_I2 `BoardIndex`: Index of digital I/O board.
- VT_I2 `OutputNumber`: Number of output to be set.
- VT_BOOL `NewState`: New state value.

`SetDigitalOutputWord`

Sets state of all (16) digital outputs of one board. Returns success or failure of set operation.
Return type: VT_BOOL
Arguments:
- VT_I2 `BoardIndex`: Index of digital I/O board.
- VT_I4 `BitMask`: Bit mask for outputs to be changed.
- VT_I4 `BitStates`: State values to be set.

`SetFieldBusOutputBit`

Sets state of an output bit of a field bus device. Returns success or failure of set operation.
Return type: VT_BOOL
Arguments:
- VT_I2 `BoardIndex`: Index of field bus board.
- VT_I2 `OutputNumber`: Number of output bit to be set.
- VT_BOOL `NewState`: New state value.

Wrapper

```
class INCApplication : public COleDispatchDriver
```

CheckRoutine Object

Property	Type, Access, Description
`ActiveScreenLayout`	`VT_I2, rw`; reads/sets screen layout of automatic screen.
`ActiveScreenLayoutName`	`VT_BSTR, ro`; returns designation of screen layout.
`Application`	`VT_DISPATCH, ro`; returns the application object.
`Author`	`VT_BSTR, rw`; reads/sets author of the check routine.
`Comments`	`VT_BSTR, rw`; reads/sets additional description of the check routine.
`Count`	`VT_I2, ro`; returns number of checks in the check routine.
`CurrentCheckResult`	`VT_BOOL, ro`; returns result of most recent execution of the given individual check. Takes one argument: - `VT_I2 SCIndex`: Number of the individual check.
`CurrentResult`	`VT_BOOL, ro`; returns final result of most recent execution of the complete check routine.
`FileName`	`VT_BSTR, ro`; returns filename of the check routine, not including path.
`FullName`	`VT_BSTR, ro`; returns full path of check routine file.
`Heading`	`VT_BSTR, rw`; reads/sets user-defined name of check routine.
`Name`	`VT_BSTR, ro`; returns filename of the check routine, not including path.
`OID`	`VT_I4, rw`; reads/sets the object's identification number (OID).
`Parent`	`VT_DISPATCH, ro`; returns the application object.
`PartsCheckedNOk`	`VT_I4`; returns total number of parts checked as "Part not O.K.".
`PartsCheckedOk`	`VT_I4`; returns total number of parts checked as "Part O.K.".
`Path`	`VT_BSTR, ro`; returns path specification for the check routine, not including filename or filename extension.
`Saved`	`VT_BOOL, ro`; returns `TRUE` if the check routine has not been changed sinced it was last saved.
`Visible`	`VT_BOOL, rw`; reads/sets visibility of the application to the user.

Method	Description
Save	Saves check routine to the file given in FullName; returns success or failure. Return type: VT_BOOL Arguments: None
SaveAs	Saves check routine to the specified file; returns success or failure. Return type: VT_BOOL Arguments: VT_BSTR FileName: name of the new file, path optional.

Collection Property	Type, Access, Description
_NewEnum	VT_DISPATCH, ro; returns an enumerator object that implements IEnumVARIANT.

Collection Method	Description
Item	Returns the given individual check object Return type: VT_DISPATCH Arguments: - VT_VARIANT SCIndex: individal check object to be returned.

Wrapper

```
class ICheckRoutine : public COleDispatchDriver
```

SingleCheck object

Property	Type, Access, Description
CheckEnabled	VT_BOOL, rw; reads/sets current activation state of the check.
Count	VT_I2, ro; returns number of check functions in the check.
CurrentResult	VT_BOOL, ro; returns final result of most recent execution of the single check.
Description	VT_BSTR, rw; reads/sets description text of the check.
LastFunction	VT_I2, ro; returns index of the last check function of the check which returned "O.K.".
Name	VT_BSTR, rw; reads/sets the name of the check.
NumOfImages	VT_I2, ro; returns number of images on the internal stack.
OID	VT_I4, rw; reads/sets the object's identification number (OID).

`Parent`	`VT_DISPATCH, ro`; returns the check routine object.
`SCIndex`	`VT_I2, ro`; returns index of single check in check routine collection.

Method	Description
`CopyImageToClipboard`	Copies an image from the data stack to the clipboard. Return type: VT_BOOL Arguments: - VT_I2 `ImageIndex`: number of the image on the stack. - VT_I2 `ImageFormat`: format of image. - VT_I2 `ImageScale`: scale factor of image.
`EnableCheck`	Enables or disables execution of the check; returns previous state (deprecated, use `CheckEnabled` property). Return type: VT_BOOL Arguments: - VT_BOOL `bEnable`: if `TRUE`, execution will be enabled.
`GetImageData`	Returns safearray containing gray level data of an image. Return type: VT_VARIANT Arguments: - VT_I2 `ImageIndex`: number of the image on the stack.
`GetImageProp`	Returns safearray containing properties of an image. Return type: VT_VARIANT Arguments: - VT_I2 `ImageIndex`: number of the image on the stack.
`IsCheckEnabled`	Returns current activation state of the check (deprecated, use `CheckEnabled` property). Return type: VT_BOOL Arguments: None

Collection Property	Type, Access, Description
`_NewEnum`	`VT_DISPATCH, ro`; returns an enumerator object that implements `IEnumVARIANT`.

Collection Method	Description
`Item`	Returns the given check function object. Return type: VT_DISPATCH Arguments: - VT_VARIANT `CFIndex`: check function object to be returned.

Wrapper

```
class ISingleCheck : public COleDispatchDriver
```

CheckFunction object

Property	Type, Access, Description
`Activated`	`VT_BOOL, rw`; reads/sets activation status of check function.
`Category`	`VT_I2, ro`; returns category of the check function.
`CFIndex`	`VT_I2, ro`; returns index of function in individual check collection.
`ColsOfParameterMatrix`	`VT_I2, ro`; returns number of columns of parameter matrix.
`ColsOfResultMatrix`	`VT_I2, ro`; returns number of columns of result matrix.
`ColsOfTargetValueMatrix`	`VT_I2, ro`; returns number of columns of target value matrix.
`ErrorCode`	`VT_I2, ro`; returns the function’s execution status after most recent execution of individual check.
`FunctionId`	`VT_I2, ro`; returns the function’s identification number.
`Name`	`VT_BSTR, rw`; reads/sets the function’s name.
`OID`	`VT_I4, rw`; reads/sets the object's identification number (OID).
`Parent`	`VT_DISPATCH, ro`; returns the single check object.
`RowsOfParameterMatrix`	`VT_I2, ro`; returns number of rows of parameter matrix.
`RowsOfResultMatrix`	`VT_I2, ro`; returns number of rows of result matrix.
`RowsOfTargetValueMatrix`	`VT_I2, ro`; returns number of rows of target value matrix.

Method	Description
`GetCurrentResult`	Returns safearray containing the result values of the check function; requires at least one execution in automatic mode. Return type: VT_VARIANT Arguments: None
`GetParameterItem`	Returns an element of the check function’s parameter matrix. Return type: VT_VARIANT Arguments: - VT_I2 `Row`: Row index of element in parameter matrix.

	- VT_I2 `Col`: Column index of element in parameter matrix.
`GetParameters`	Returns safearray containing the current parameter settings of the check function. Return type: VT_VARIANT Arguments: None
`GetResultItem`	Returns an element of the check function's current result matrix; requires at least one execution in automatic mode. Return type: VT_VARIANT Arguments: - VT_I2 `Row`: Row index of element in result matrix. - VT_I2 `Col`: Column index of element in result matrix.
`GetTargetValueItem`	Returns an element of the check function's target value matrix. Return type: VT_VARIANT Arguments: - VT_I2 `Row`: Row index of element in target value matrix. - VT_I2 `Col`: Column index of element in target value matrix.
`GetTargetValues`	Returns safearray containing the current target value settings of the check function. Return type: VT_VARIANT Arguments: None
`HasCurrentResult`	Returns `TRUE` if the check function returns result values to the controller. Return type: VT_BOOL Arguments: None
`HasParameters`	Returns `TRUE` if the check function makes its parameter settings available to the controller. Return type: VT_BOOL Arguments: None
`HasTargetValues`	Returns `TRUE` if the check function makes its target value settings available to the controller. Return type: VT_BOOL Arguments: None
`SetParameterItem`	Sets an element of the check function's parameter matrix. Returns success or failure of operation. Return type: VT_BOOL Arguments: - VT_I2 `Row`: Row index of element in parameter matrix. - VT_I2 `Col`: Column index of element in parameter matrix. - VT_VARIANT `NewValue`: New value of specified element.

`SetParameters`	Sets parameters of the check function. Returns success or failure of operation. Return type: VT_BOOL Arguments: - VT_VARIANT `ParaSetting`: safearray containing parameters.
`SetTargetValueItem`	Sets an element of the check function's target value matrix. Returns success or failure of operation. Return type: VT_BOOL Arguments: - VT_I2 `Row`: Row index of element in target value matrix. - VT_I2 `Col`: Column index of element in target value matrix. - VT_VARIANT `NewValue`: New value of specified element.
`SetTargetValues`	Sets target values of the check function. Returns success or failure of operation. Return type: VT_BOOL Arguments: - VT_VARIANT `TargetSetting`: safearray containing target values.

Wrapper

```
class ICheckFunction : public COleDispatchDriver
```

Check Functions with Additional Automation Functionality

Identify Bar Code

Function ID: `BCI=536`

Functionality	Description
Parameter values	- Code type: value indicating type of bar code. - Line distance: distance of search rays. - Smoothing: smoothing parameter. - Scan direction: value indicating scan direction. - Check sum: if `TRUE`, the function performs a check sum test. - Characters: number of charactes contained in the bar code.
Target values	- Check target code: if `TRUE`, code is compared to target string. - Target string: bar code string to be present on the part.
Result values	- Bar code string: string with the identified bar code.

Identify DataMatrix Code

Function ID: `DMCI=552`

Functionality	Description
Parameter values	- Code type: value indicating type of DataMatrix code. - Code color: value indicating color of code (dark or light). - Code quality: value indicating quality of code (good or poor). - Code size: approximate code size in pixels. - Reference: value indicating reference angle. - Range: range of angle. - Undersampling: sub sampling parameter. - Minimum edge height: contrast required for edges.
Target values	- Check target code: if `TRUE`, code is compared to target string. - Target string: bar code string to be present on the part.
Result values	- DataMatrix code string: string with the identified code.

Count ROIs

Function ID: `OBC=510`

Functionality	Description
Target values	- Check count: if `TRUE`, ROI count of first group is verified. - Minimum: minimum number of ROIs required in first group. - Maximum: maximum number of ROIs allowed in first group.
Result values	- Count: current number of ROIs in first group.

Evaluate Classes

Function ID: `CLE=543`

Functionality	Description
Target Values	- Verify: if `TRUE`, function will verify the classes of ROIs. - Rejection threshold: minimum certainty required for "O.K.". - Class strings: up to 20 class names for verification.
Result values	Results are returned in an array of structures holding the following values for each ROI: - Class string: name of the identified class - Quality: classification certainty

Check Allowances

Function ID: `GCHK=527`

Functionality	Description
Target values	Target values are set or returned in an array of structures holding the following values for each measurement: - Verify: if `TRUE`, measurement will be compared. - Description: name of the measurement (read-only) - Nominal value: nominal value of the measurement - Lower allowance: determines lower limit of the measurement - Upper allowance: determines upper limit of the measurement
Result values	Result values are returned in an array of structures holding the following values for each measurement: - Description: name of the measurement - Current value: current value of the measurement

Copy ROIs

Function ID: `OCPY=525`

Functionality	Description
Result values	Result values are returned in an array of structures holding the following values for each ROI and each feature: - Object number: identifier of ROI - Feature ID: identifier of feature - Current value: value of the feature

Determine Position

Function ID: `POSC=521`

Functionality	Description
Result values	- X Offset: offset in x direction to reference point - Y Offset: offset in y direction to reference point - Rotation: rotation angle relative to reference image - Pivot X: x coordinate of current pivot point - Pivot Y: y coordinate of current pivot point

Capture Image

Function ID: `DIG=517`

Functionality	Description
Parameter values	- Camera: identifier (index) of camera

Transfer Image

Function ID: `IDT=508`

Functionality	Description
Parameter values	- Left: X coordinate of top left corner - Top: Y coordinate of top left corner - Right: X coordinate of bottom right corner - Bottom: Y coordinate of bottom right corner - Source: identifier of image source - Camera: identifier (index) of camera - Bitmap: name of bitmap file - Tray index: index of image tray

Determine Threshold

Function ID: `ITH=506`

Functionality	Description
Parameter values	- Use manual threshold: if `TRUE`, manual threshold is used - Manual threshold: value of the threshold - Result image: parameter for automatic threshold computation to adjust predominance of light or dark areas - Defect suppression: parameter for automatic threshold computation to suppress disturbances

Define ROIs

Function ID: `DAOI=512`

Functionality	Description
Parameter values	- Left: X coordinate of top left corner - Top: Y coordinate of top left corner - Right: X coordinate of bottom right corner - Bottom: Y coordinate of bottom right corner

Classify ROIs

Function ID: `OCL=518`

Functionality	Description
Parameter values	- Classifier: name of the classifier attached to this function.

Screen ROIs

Function ID: `OBF=513`

Functionality	Description
Parameter values	Parameter values are set and returned in an array of structures holding the following values for each feature: - Verify: if `TRUE`, feature is activated for screening - Feature ID: identifier of current feature (read-only) - Minimum: minimum value allowed for the current feature - Maximum: maximum value allowed for the current feature

Rotate Image

Function ID: `IROT=507`

Functionality	Description
Parameter values	- Mode: value indicating rotation angle or mirror direction

Template Matching

Function ID: `TMA=546`

Functionality	Description
Parameter values	- Result positions: number of objects the function will create at most in first group. - Minimum quality: required degree of correspondence for first group.

Check Function Categories

The check function categories are defined in `ncauto.h`.

Value	Symbol and Meaning	
0	`NC_FCTCAT_GRAYVALUE`	preprocessing, gray level
1	`NC_FCTCAT_IMGFILTER`	preprocessing, filters
2	`NC_FCTCAT_IMGGEO`	preprocessing, geometry
3	`NC_FCTCAT_COLORANALYSIS`	color analysis
4	`NC_FCTCAT_OBJCREATE`	image analysis, object creation
5	`NC_FCTCAT_OBJFEATURES`	image analysis, object features
6	`NC_FCTCAT_OBJEVALUATION`	image analysis, object evaluation
7	`NC_FCTCAT_TOOLS`	tools

8	`NC_FCTCAT_IMGACQ`	image acquisition
9	`NC_FCTCAT_POSADJUSTMENT`	position adjustment
10	`NC_FCTCAT_GAUGING`	gauging
11	`NC_FCTCAT_PLUGIN`	plug-in functions
12	`NC_FCTCAT_IOCOM`	IO communication

Check Function IDs

Value	Mnemonic	Meaning
500	INRM	Enhance Image (**i**mage **n**o**r**m**m**alize)
501	DLEX	**Del**ay **Ex**ecution
502	ICPY	**C**o**py** **I**mage
503	ICMB	**C**o**mb**ine **I**mage
504	ILUT	Apply **L**ook **u**p **T**able to **I**mage
505	IFIL	**Fil**ter **I**mage
506	ITH	Determine **Th**reshold
507	IROT	**Rot**ate **I**mage
508	IDT	Transfer Image (**I**mage **D**ata **T**ransfer)
509	OBIN	Create **O**bjects by Thresholding (**bin**ary)
510	OBC	**C**ount ROIs (**ob**jects)
511	OBS	**S**ort ROIs (**ob**jects)
512	DAOI	**D**efine Regions (**A**reas) **o**f **I**nterest
513	OBF	Screen ROIs (**ob**ject **f**iltering)
514	SOU	**S**et Digital **Ou**tput
515	RIN	**R**ead Digital **In**put
517	DIG	Capture Image (**dig**itize)
518	OCL	**Cl**assify ROIs (**o**bjects)
519	OBM	Compute Features (**ob**ject **m**easurements)
520	OBG	Resample ROIs (**ob**ject on **g**rid)
521	POSC	Determine Position (**pos**ition **c**alculation)

523	POSR	**Pos**ition **R**OIs
524	PCA	**Cal**ibrate **P**ixels
525	OCPY	**Copy** ROIs (**o**bjects)
526	DGEO	Gauge ROIs (**d**efine **geo**metrical measurements)
527	GCHK	**Ch**ec**k** Allowances (**g**eometry)
528	GCMB	Derive Measurements (**g**eometry **comb**ine)
529	DIGA	Capture Image in Parallel (**dig**itize **a**synchronously)
531	ISHA	**Sha**ding Correction (of **i**mage)
532	OCMG	**C**ompute **M**odel **G**eometries (for **o**bjects)
536	BCI	**I**dentify **B**ar **C**ode
537	OUR	**U**nroll ROI (**ob**ject)
538	GCB	**C**ali**b**rate measurements (**g**eometrical)
539	OMG	Combine ROIs (**m**er**g**e **o**bjects)
540	GMG	Combine measurement lists (**m**er**g**e **g**eometries)
541	ISM	**Sm**ooth ROIs (in **i**mage)
542	OBCV	Compute **C**ur**v**ature (of **ob**jects)
543	CLE	**E**valuate **Cl**asses
544	IFI	Transfer **I**mage to Tray (**fi**ling)
545	ODR	**Dr**aw ROIs (**o**bjects)
546	TMA	**T**emplate **Ma**tching
547	CMA	**C**olor **Ma**tching
548	PQI	**P**rint **Q**uality **I**nspection
549	OBED	Create **Ed**ges (as **ob**jects)
550	RFBB	**R**ead **F**ield **B**us Input (**B**it)
551	SFBB	**S**et **F**ield **B**us Output (**B**it)
552	DMCI	**I**dentify **D**ata**M**atrix **C**ode
553	OCT	Split ROIs (**o**bject **c**u**t**)
554	IOFIL	**Fil**ter **I**mage in ROIs (**o**bject)
555	ISL	Adjust Line-Scan-Image (**i**mage **s**hift **l**ines)

556	CIA	**C**ontrol **I**mage **A**cquisition

Error Type IDs

Value	Meaning
0	General Failure.
1	Set property failed.
2	Get property failed.
3	Method failed.
4	Automation failure.

Error Detail IDs

Value	Meaning
1	General failure.
2	Not available for Runtime version.
3	Not available for Demo version.
4	Only available in Live mode.
5	Only available in Automatic mode.
6	NCheck not configured for OLE automation.
7	No check routine loaded.
8	Device not available/not configured.
9	Index out of range.
10	Index of device out of range.
11	Index of input/output bit out of range.
12	Row index out of range.
13	Column index out of range.
14	Unknown value for constant.
15	Check function has no target values.
16	Check function has no parameters.

17	Check function has no result.
18	Input structure invalid.
19	Input type invalid.
20	Could not set input value.
21	Input value invalid.
22	Element is read-only.
23	File not found.
24	Cannot save check routine protected by password.
25	Cannot overwrite check routine protected by password.
26	Path not found.
27	Invalid dimensions for application window .
28	Not available under current security settings.
29	General failure for accessing device.
30	Error in SafeArray operation.
31	Invalid structure for SafeArray setting.
32	Warning: at least one check routine is configured for image transfer from "File".
33	Cannot read file created by demo version.
34	Check routine was created using a plug-in extension currently unavailable.
35	Property / method no longer supported.
36	Access denied under current security settings.
37	Parameters not available for polyline.
38	Input value was empty.
39	Check function must be initialized in NeuroCheck.
40	Unexpected file format.
41	Check function cannot be deactivated.