ARGES ControllerLib 2023-09-18

User Manual (Original)

Application Programming Interface
for ARGES System Controllers

Read carefully before using.
Retain for future reference.

Novanta
PHOTONICS

Manufacturer

Novanta Europe GmbH
Werk 4, 92442 Wackersdorf, Germany
Phone: +49-9431-7984-0

Email: TeamArges@Novanta.com

Customer support

Before contacting us for assistance, review appropriate sections in this manual that may answer your questions. After consulting the manual, please contact one of our worldwide offices between 9 AM and 5 PM local time.

Americas, Asia Pacific

Novanta Headquarters
Bedford, USA
Phone: +1-781-266-5700

Email: Photonics@Novanta.com

Europe, Middle East, Africa

Novanta Europe GmbH
Wackersdorf, Germany
Phone: +49-9431-7984-0

Milan, Italy
Phone: +39-039-793-710

Email: Photonics@Novanta.com

China

Novanta Sales & Service Office
Shenzhen, China
Phone: +86-755-8280-5395

Suzhou, China
Phone: +86-512-6283-7080

Email: Photonics.China@Novanta.com

Japan

Novanta Service & Sales Office
Tokyo, Japan
Phone: +81-3-5753-2460

Email: Photonics.Japan@Novanta.com

Be mindful of the environment,
do not print if you do not need to.

This PDF file forces duplex printing,
if your printer can do that.

Table of contents

1. How to use this document **1**

2. The interface and concept in brief **5**

3. Usage of the interface **6**
3.1. Installing 6
3.2. Writing, compiling and linking the code 7
3.3. Distributing your software application 8
3.4. Updating 9
3.5. Uninstalling 9
3.6. Troubleshooting 10

4. Syntax of the interface **11**
4.1. Typedefs 11
4.1.1. Terms 11
4.1.2. Predefined typedefs 12
4.2. Callback functions 22
4.2.1. Nodes 22
4.2.2. PLC changes 35
4.2.3. System messages 37
4.2.4. Devices 39
4.2.5. Error, RequestLog, Log, JobLines 42
4.3. Supported states 44
4.3.1. NodeObject states 44
4.3.2. Device states 45
4.3.3. Device error states 45
4.3.4. Device parameter states 46
4.3.5. Device power states 46
4.3.6. PLC states 47
4.4. Constants 47
4.4.1. Error codes 47
4.4.2. Variable types 48
4.4.3. PLC states 48
4.5. Functions 50
4.5.1. General 50

4.5.2. Controller handling 56
4.5.3. Error handling 66
4.5.4. Requests 68
4.5.5. Paths 73
4.5.6. Files 76
4.5.7. Nodes 91
4.5.8. NodeObjectCollections 163
4.5.9. Jobs 172
4.5.10. Drivers 186
4.5.11. Devices 188
4.5.12. Callbacks 202
4.5.13. System messages 264
4.5.14. (Scan) heads 276
4.5.15. Distortion and scan field correction 277
4.5.16. Logging 279

5. Semantics of the interface 285

5.1. Working with jobs 286
5.1.1. Getting a list of jobs 286
5.1.2. Loading a job 286
5.2. Working with NodeObjects 287
5.2.1. Reading variable values 288
5.2.2. Writing variable values 289
5.2.3. Working with value change callbacks 289
5.2.4. Creating user variables 289
5.2.5. Getting information about NodeObjects 290
5.3. Working with NodeObjectCollections 290
5.3.1. Creating and destroying NodeObjectCollections 290
5.3.2. Using NodeObjectCollections 291
5.4. Using the VariableCache 291
5.5. Source code examples 292
5.5.1. Linking and compiling an example 293
5.5.2. Reading a variable 293
5.5.3. Writing a variable 296
5.5.4. Watching a variable 299
5.5.5. Loading a job 302
5.5.6. Marking a job 304

A. Copyright and licenses **309**
A.1. Proprietary software 309
A.2. Open source software 309

B. License texts **311**
B.1. FreeType Project License 311
B.2. ImageMagick License 315
B.3. InScript License 318
B.4. Zlib License 322

C. Trademarks **324**

D. Bibliography **325**
Opening the InScript and Firmware User Manual on-screen 325

E. Acronyms **327**

Index **328**

1. How to use this document

Symbols and signal words

This manual uses the following symbols and signal words for information of importance.

⚠ NOTICE

... indicates a hazardous situation which is not related to personal injury and which *could* result in *material damage*, if not followed.

IMPORTANT

... indicates information considered important but not directly hazard related.

TIP

... indicates a practice which facilitates work.

Procedure

... indicates a call to action. One or more steps to be executed are following this signal word. An exception has been made for warning notices, where this signal word has been omitted.

fct

```
… shows the function syntax
```

If the function communicates with the firmware on the controller then this takes more time than usually. This is indicated by a clock symbol ◷ in the box.

par ... lists the function parameters

ret ... lists the function returns

see ... cross references to related functions

Notation conventions

This manual uses the following notation conventions that are given in the following table.

Table: Notation conventions

Notation	Meaning
Notation in body text:	
Italic	Emphasized text
`Typewriter`	File name or path; functions, parameters and returns
Underlined	Click-able cross reference or hyper link (only visible on-screen)
[⟨Number⟩]	Cross reference to the bibliography on page 325
Notation in procedures:	
Bold	Element in the graphical user interface that the user shall click
Italic	Name of element in the graphical user interface
`Typewriter`	Text to be typed in by the user or file name or path
Notation in program listings:	
`Blue`	Keywords
`Gray`	Comments
`Green`	Preprocessor commands
`Red`	Strings

Notation	Meaning
®	... indicates a registered trademark; see also page 324. We omitted this notation for our own registered trademarks ARGES® and InScript® to improve readability

Storage and replacement

Novanta reserves the right to update this manual at any time without prior notification.

A replacement for this manual can be requested. The manual is available as a PDF-file.

Procedure

- Keep this manual with the controller to access it at any time during the controller's lifetime.
- This manual is part of the product. If the product ownership changes then this manual must accompany the product.

Target audience, specifics and structure

This document addresses software developers.

For applications in automation most software developers favor the following approach to use the ARGES ControllerLib. They set up devices, pens, jobs, and fonts in the InScript software first and then use their own software application, they have developed with the ARGES ControllerLib, to control the functionality of the ARGES system controller.

It would indeed be possible to develop a software application that also covers the functionality necessary for the initial steps but this would not be cost-effective as this approach ignores functionality already at hand with the InScript software. Also keep in mind that the ARGES ControllerLib is just a *low-level* application programming interface and that you would have to develop the necessary functionality from scratch.

With this general approach in mind we structured the document as follows.

Chapter 2 describes in brief the interface and its concept.

Chapter 3 describes how to integrate the ARGES InScript software package with custom applications by using the ARGES ControllerLib.

Chapter 4 describes the means to write code that drives the ARGES system controller. This chapter is a comprehensive reference to those means, namely typedefs, callback functions, supported states, constants and functions. As this chapter is the better part of the whole document there are 2 provisions that keep the main table of contents short, hence clear, and still provide means to find what you are searching for. Each subsection in this chapter holds a local table of contents. And an additional alphabetical Index is located at the end of this document.

Chapter 5 describes in basic examples how to write code that drives the ARGES system controller.

2. The interface and concept in brief

The ARGES ControllerLib is a low-level application programming interface for the ARGES system controller. It is a C-based library and DLL that gives you a collection of procedures and functions at hand to control the functionality of the ARGES system controller from within your own software application.

On an ARGES system controller all is represented by so called NodeObjects. There are different types of nodes. Each type has a function and properties. The nodes are arranged in a hierarchical tree structure where child nodes may inherit properties of their parent nodes. Amongst others there are nodes for devices, pens and jobs:

- A device is the software representation of ARGES system controller hardware or hardware connected to that controller.
- A pen defines parameters for 1 or more devices. Pens are used to set device parameters during job execution, e.g. the laser power, frequency or processing speed.
- A job is a tree structure of nodes that control devices. Jobs are used to define the workflow of processing.

The ARGES ControllerLib handles all aspects of these items.

3. Usage of the interface

This chapter covers the life-cycle procedures of the ARGES ControllerLib from installation through operation and updating to uninstallation.

The chapter also covers troubleshooting regarding these life-cycle procedures. This document however does not cover debugging.

3.1. Installing

Requirements

- ☐ PC for software development (obviously) with one of these operating systems:
- ☐ Windows® 7 / 8 / 10 (32-bit and 64-bit variants)
  – OR –
  macOS® 10.6 / 10.7 / 10.8
  – OR –
  A multitude of Linux® variants and operating systems that are not listed here

  Please, contact ARGES for more information about the latter item.

  Typically the operating system is the same as that one you want to develop your software application for.
- ☐ on Windows®: Visual Studio®
  – OR –
  on Linux®: GNU C/C++ compiler

- ☐ ARGES system controller that is connected by Ethernet to said PC

  For information about how to connect the ARGES system controller to a PC see [1].

Procedure

1. Unpack file
   `controllerlib_<operating system>_<version>.zip` or
   `controllerlib_<operating system>_<version>.tgz` to a convenient path.
2. Add this path to the *include path* and *library path* of your project.

3.2. Writing, compiling and linking the code

Requirements

- ☐ Installed ARGES ControllerLib
- ☐ The following procedure applies when running the GNU C/C++ compiler on Linux®.

  If you are running Visual Studio® on Windows® however then this is not described here but the procedure applies essentially.

Procedure

1. Write your ARGES ControllerLib code by the means given in chapters 4 and 5.

   Once you have written your code, compile it and link it to the ARGES ControllerLib.
2. In the command line type

```
g++ -o <software> -lcontrollerlib -lpthread <code>.c
```

   where

   `<software>` is the filename of your software

<code> is the filename of the ARGES ControllerLib code that you have written

3. Press the **ENTER**-key.

   If errors occur then debug the code. This document, as already mentioned, does not cover debugging.

   If no error occurs then this results in an executable software application that will work on the PC that you are using for software development as you already have added the path of the ARGES ControllerLib's `*.dll` files to the *include path* and *library path* of your project during the installation procedure. Hence, the `*.dll` files are "known" to this PC.

   For distribution of your software application however you have to include these `*.dll` files into the installation routine of your software application.

3.3. Distributing your software application

⚠ NOTICE

Infringing copyrights, licenses or trademarks
may result in financial losses

- Make sure not to infringe copyrights and licenses and trademarks of the ARGES ControllerLib, see appendices A and C, as well as of your own part of the software application.

To make your software application work on a PC other than the PC that you are using for software development, all `*.dll` files from the folder of the ARGES ControllerLib installation are needed.

Procedure

- From the folder of the ARGES ControllerLib installation include all `*.dll` files into the installation routine of your software application and make the installation wizard copy these files into the installation folder of your software application.

3.4. Updating

Requirements

- ☐ Installed ARGES ControllerLib

Procedure

1. Optionally save the folder of the ARGES ControllerLib installation and its content as an “Angst”-copy.
2. In the folder of the ARGES ControllerLib installation delete all files and folders.
3. Unpack the contents of the new file `controllerlib_<operating system>_<version>.zip` or `controllerlib_<operating system>_<version>.tgz` to the folder of the ARGES ControllerLib installation.

   During the installation procedure you have already added the path of the ARGES ControllerLib’s `*.dll` files to the *include path* and *library path* of your project.

3.5. Uninstalling

Requirements

- ☐ Installed ARGES ControllerLib

Procedure

1. Delete the folder of the ARGES ControllerLib installation.
2. Remove the path to the folder of the ARGES ControllerLib installation from the *include path* and *library path* of your project.

3.6. Troubleshooting

The section covers troubleshooting regarding the life-cycle procedures that are described above. This document, as already mentioned, does not cover debugging.

Table 3.1.: Troubleshooting

Symptom	Cause	Measure
[to be defined]	[to be defined]	[to be defined]
[to be defined]	[to be defined]	[to be defined]

4. Syntax of the interface

4.1. Typedefs

4.1.1. Terms

A typedef declaration is used within its scope to give a data type a new name.

HController is the handle to the controller. With this handle the controller can be identified

HNodeObject is the handle to the NodeObject. With this handle the NodeObject can be identified

HNodeObjectCollection is the handle to the NodeObjectCollection. With this handle the NodeObjectCollection can be identified

HSysMsg is the handle to the SysMsg. With this handle the SysMsg can be identified

HHead is the handle to the (scan) head. With this handle the (scan) head can be identified. This handle is used for distortion or scanfield correction

HScanfieldCorrection is the handle to the scanfield correction

4.1.2. Predefined typedefs

Table of contents

4.1.2.1. ARG_NODEINFO 12
4.1.2.2. ARG_JOBNODEINFO 14
4.1.2.3. ARG_JOBINFO 14
4.1.2.4. ARG_LINEINFO 15
4.1.2.5. ARG_DRIVERINFO 16
4.1.2.6. ARG_SINGLEDRIVERINFO 17
4.1.2.7. ARG_DEVICESTATE 17
4.1.2.8. ARG_DEVICEINFO 18
4.1.2.9. ARG_SINGLEDEVICEINFO 18
4.1.2.10. ARG_FEATURELIST 19
4.1.2.11. ARG_FEATURE 19
4.1.2.12. ARG_TSS_CHANNELTYPE 20
4.1.2.13. ARG_TSS_INFO 20
4.1.2.14. ARG_TSS_DATA 20

4.1.2.1. ARG_NODEINFO

Used with 4.5.7.3 GetNodeInfo to get the information about a node. All strings are valid and have an empty string if not set. This structure contains the following fields:

HNodeObject handle is the handle to the NodeObject

int64 nodeid is the ID of the NodeObject on the controller. This equals a call to 4.5.7.40 GetNodeID

char *fullname is the full name of the NodeObject; e.g. "`stat.time.TimeStr`". This equals a call to 4.5.7.42 GetNodeName

char *name is only the variable name of the NodeObject; e.g. "`TimeStr`". Please note that it is possible to have NodeObjects with an empty name

char *path is the path of the NodeObject; e.g. "`stat.time`"

char *value is the value of the NodeObject as string (if possible). This equals a call to 4.5.7.82 GetNodeValueString

char *priv are the options for VAR:SEL-NodeObjects which are seperated by `0x0D`. Normally the priv-part of variables is used for VAR:SELECT-nodes. To extract the select-entries it might be more comfortable to call 4.5.7.73 GetSelectEntriesCount and 4.5.7.75 GetSelectEntry

char *unit is the unit for the NodeObject. This equals a call to 4.5.7.95 GetUnit

int type is the type of the NodeObject. This equals a call to 4.5.7.37 GetNodeType. This type is only valid on nodes which are not job nodes

char *typestring is the type of the NodeObject as string; e.g. "`VAR:SET`", "`JOB:EXEC`", ... This equals a call to 4.5.7.39 GetNodeTypeString

float min is the minimum value of the NodeObject. If "`min`" is greater than "`max`" there are no restrictions. This equals a call to 4.5.7.92 GetMin

float max is the minimum value of the NodeObject. If "`min`" is greater than "`max`" there are no restrictions. This equals a call to 4.5.7.93 GetMax

unsigned int index is the index of the NodeObject. The index is used to evaluate the order of nodes on a given level in the tree. This equals a call to 4.5.7.36 GetNodeIndex

unsigned int flags are the flags of the NodeObject

Listing 4.1: ARG_NODEINFO example

```
/* Get the Handle for a NodeObject */
HNodeObject HNO = GetNode(HC, varname);

ARG_NODEINFO *nodeinfo = GetNodeInfo(HC, HNO);
if ( nodeinfo ) {
  printf("Fullname: %s\n", nodeinfo->fullname);
  printf("Lastname: %s\n", nodeinfo->name);
  printf("Value:    %s\n", nodeinfo->value);
  printf("Unit:     %s\n", nodeinfo->unit);
  printf("Type:     %s\n", nodeinfo->typestring);
```

```
    DestroyNodeInfo(nodeinfo);
  } else {
    printf("Node %s not found.\n",varname);
  }
```

4.1.2.2. ARG_JOBNODEINFO

Used with 4.5.9.7 GetJobNodeTypeInfo to get the information for a job node type. All strings are valid and have an empty string if not set. This structure contains the following fields:

char *fullname is the full name of the Job node type, e.g. "`JOB:SHAPE`". This name should be used to create job nodes on the controller

char *name is the name of the Job node type, e.g. "`SHAPE`"

int usercreatable `1`: if the job node can be created by the user (with 4.5.7.30 CreateNodeOnController); `0`: otherwise

int canmakelines `1`: if the job node produces output lines when executed; `0`: otherwise

int canpasslines `1`: if the job node passes output lines to sub-nodes when executed; `0`: otherwise

int acceptssubnodes `1`: if the job node can have job nodes as children; `0`: if not

4.1.2.3. ARG_JOBINFO

Used with 4.5.9.1 GetJobInfo to get the information for a job node type. All strings are valid and have a empty string if not set. This structure contains the following fields:

int jobcount is the count of jobs on the controller. The string arrays `fulljobnames` and `jobnames` are filled with the names from `0` to `jobcount-1`. If there are no jobs on the controller the `fulljobnames` and `jobnames` are undefined

char **fulljobnames are the full names of the jobs; e.g.
`jobnames[0] = ”usr.job.MyJob”, jobnames[1] = ”usr.job.MyJob1”`

char **jobnames are only the last parts of names; e.g.
`jobnames[0] = ”MyJob”, jobnames[1] = ”MyJob1”`

char *selectednode contains the full name of the selected node,
e.g. “`usr.job.Job`”

> **TIP**
>
> Not only jobs can be selected but also any other job node like “`usr.job.MyJob.Shape1`”.

4.1.2.4. ARG_LINEINFO

Used with 4.5.9.9 GetJobLines to get information about the lines of a job.

> **TIP**
>
> This information is used in InScript to show a preview of a job.

This structure contains the following fields:

HNodeObject HNO is the handle to the NodeObject for which the linedata is valid

int includessubnodes `1`: if sub-nodes in the line data are included, `0`: otherwise

int packagenumber is for each 4.5.9.9 GetJobLines-call the callback function that can be called several times. The package number indicates which package number it is (starting at `1`)

int lastpackage `1`: if this is the last package for the 4.5.9.9 GetJobLines-call; `0`: otherwise

DWORD userhandle is the handle that the user has set with 4.5.9.9 GetJobLines

unsigned int datalength is the length of the data in bytes

void *data is the line data. It is ensured that every package has a complete set of valid line data.

ARL is an efficient vector graphics file format specified by ARGES. It is the recommended file format for 2D and 3D vector data in InScript. If you want to create ARL files with your own software then please contact our service department to get the file format description

4.1.2.5. ARG_DRIVERINFO

Used with 4.5.10.1 GetDriverInfo to get the information about the drivers available with the firmware in use on the controller.

int drivercount is the number of available drivers

ARG_SINGLEDRIVERINFO **driver is the pointer to an array of ARG_SINGLEDRIVERINFO-structures; see 4.1.2.6 ARG_SINGLEDRIVERINFO

Listing 4.2: ARG_DRIVERINFO example

```
ARG_DRIVERINFO *info = GetDriverInfo(HC);
if ( info != NULL ) {
  printf("\n\n");
  printf("Found %i drivers\n", info->drivercount);
  for (int i=0; i<info->drivercount; ++i) {
    ARG_SINGLEDRIVERINFO *driverinfo = info->driver[i];
    printf("Driver: %s (Version: %s)\n", driverinfo->name,
       driverinfo->version);
    printf("Vendor: %s\n", driverinfo->vendor);
    printf("Comment: %s\n", driverinfo->comment);
    printf("\n");
  }
  DestroyDriverInfo(info);
  return E_OK;
}
```

4.1.2.6. ARG_SINGLEDRIVERINFO

Used with 4.5.10.1 GetDriverInfo to get the information about the drivers available with the firmware in use on the controller. All strings are valid and have a empty string if not set. See 4.1.2.5 ARG_DRIVERINFO or 4.5.10.1 GetDriverInfo for an example of how to use this structure.

char *name contains the name of the driver; e.g. “`LINEPAR`”

char *version contains the version of the driver; e,g, “`1.18`”

char *vendor contains the vendor of the driver; e.g. “`ARGES GmbH`”

char *comment contains the comment of the driver; e.g. “`This device is for the line parameters.`”

char *classification contains the classification of the driver; e.g. “`Laser`”

4.1.2.7. ARG_DEVICESTATE

A device can have different states. Following states are possible:

- ARG_DEV_UNINITIALIZED
- ARG_DEV_INITIALIZING
- ARG_DEV_DESTROYING
- ARG_DEV_ACTIVATING
- ARG_DEV_ACTIVE
- ARG_DEV_DEACTIVATING
- ARG_DEV_INACTIVE
- ARG_DEV_ACTIVATING_FAILURE
- ARG_DEV_DEACTIVATING_FAILURE
- ARG_DEV_UNKNOWN_FAILURE

4.1.2.8. ARG_DEVICEINFO

Used with 4.5.11.7 GetDeviceInfo to get the information about the devices available with the firmware in use on the controller. See 4.5.11.7 GetDeviceInfo for an example of how to use this structure.

devicecount is the number of available devices

device is the pointer to an array of ARG_SINGLEDEVICEINFO-structures; see 4.1.2.9 ARG_SINGLEDEVICEINFO

Listing 4.3: ARG_DEVICEINFO example

```
HDevice HD = GetDevice(HC,”My Laser”);
if ( HD != ARG_INVALID_HANDLE_VALUE ) {
  ARG_SINGLEDEVICEINFO *info = GetDeviceInfo(HC, HD);
  if ( info ) {
    printf(”Devicename: %s\n”, info->name);
    DestroySingleDeviceInfo(info);
  }
}
```

4.1.2.9. ARG_SINGLEDEVICEINFO

Used with 4.5.11.7 GetDeviceInfo to get the information about the devices available with the firmware in use on the controller. All strings are valid and have a empty string if not set. See 4.1.2.8 ARG_DEVICEINFO or 4.5.11.7 GetDeviceInfo for an example of how to use this structure.

HDevice handle is the handle for the device

char *name is the name of the device

char *driver is the name of the driver

char *dependenciesvarname is the variable, where dependencies are stored

int dependenciescount is the number of dependencies

char **dependencies is each dependency as a string

int dependsoncount is the number of devices this device depends on

char **dependson is each device that this device depends on as a string

ARG_DEVICE_ACTIVATION_STATE activationstate is the current activation state of the device

ARG_DEVICE_ERROR_STATE errorstate is the current error state of the device

ARG_DEVICE_PARAM_STATE paramstate is the current parameter state of the device

ARG_DEVICE_POWER_STATE powerstate is the current power state of the device

4.1.2.10. ARG_FEATURELIST

Used with 4.5.2.3 GetFeatureList to get the information about the features available with the firmware in use on the controller.

int featurecount is the number of features

ARG_FEATURE **features is a pointer to an array of ARG_FEATURE-structures

4.1.2.11. ARG_FEATURE

Used with 4.5.2.3 GetFeatureList to get the information about the features available with the firmware in use on the controller. All strings are valid and have a empty string if not set. See 4.5.2.3 GetFeatureList for an example of how to use this structure.

char *name contains the name of the feature. For documentation on what the features mean, please consult the documentation of the firmware in use; see [2]

int version contains the version of the feature; e,g. “**1**”

int flags contains the flags of the feature. For documentation on what the flags mean, please consult the documentation of the firmware in use; see [2]

4.1.2.12. ARG_TSS_CHANNELTYPE

Used in the 4.1.2.13 ARG_TSS_INFO-structure for information about a TimedSignalStream. All strings are valid and contain an empty string if empty. For an example of how to use this structure see 4.5.9.15 GetTssInfo. This structure contains the following fields:

name is the name of the channeltype

coordname is the name of the coordinate system

axiscount is the number of axes

axisname is the name of each axis

4.1.2.13. ARG_TSS_INFO

Used with 4.5.9.15 GetTssInfo to get the information for TimedSignalStreams. For an example of how to use this structure see 4.5.9.15 GetTssInfo. This structure contains the following fields:

channelcount is the number of channels in the TimedSignalStream

ARG_TSS_CHANNELTYPE **channeltype is an array of channeltypes; see 4.1.2.12 ARG_TSS_CHANNELTYPE for information on that structure

4.1.2.14. ARG_TSS_DATA

connectionid is the ID of the TimedSignalStream connection

channelname is the name of the TimedSignalStream channel

channelspecifier is the specifier of the TimedSignalStream channel

blocknumber is the current data block number

totalblockcount is the number of total blocks; `0`: if unknown. If block number and total block count are equal then the last package has arrived

aborted is `1`: if the stream was aborted; `0`: if it was not aborted

starttimestamp is the Starttimestamp

endtimestamp is the Endtimestamp

samplespersec are the sampes per second

datalen is the length in bytes of the data

data is the data

4.2. Callback functions

4.2.1. Nodes

Table of contents

4.2.1.1. NodeModifiedCallbackFunction 23
4.2.1.2. ValueChangeCallbackFunctionExt 23
4.2.1.3. ValueChangeCallbackFunction 24
4.2.1.4. FlagsChangeCallbackFunctionExt 24
4.2.1.5. FlagsChangeCallbackFunction 25
4.2.1.6. NameChangeCallbackFunctionExt 26
4.2.1.7. NameChangeCallbackFunction 26
4.2.1.8. NodeCreatedCallbackFunctionExt 27
4.2.1.9. NodeCreatedCallbackFunction 28
4.2.1.10. StartOfNodeCreatedRequestCallbackFunction 28
4.2.1.11. EndOfNodeCreatedRequestCallbackFunction 29
4.2.1.12. NodeDeletedCallbackFunctionExt 29
4.2.1.13. NodeDeletedCallbackFunction 30
4.2.1.14. StartOfNodeDeletedRequestCallbackFunction 31
4.2.1.15. EndOfNodeDeletedRequestCallbackFunction 31
4.2.1.16. NodeWillBeDeletedCallbackFunction 32
4.2.1.17. NodeMovedCallbackFunction 32
4.2.1.18. NodeMovedCallbackFunctionExt 33
4.2.1.19. NodeStateChangeCallbackFunction 34

4.2.1.1. NodeModifiedCallbackFunction

fct

```
int (* NodeModifiedCallbackFunction(HController HC,
HNodeObject HNO, void *userpointer, int reserved));
```

Functions of this type can be registered as callback functions for Node-Modified-events (RegisterOnNodeModified).

TIP

- This callback only works correct if the variable cache was enabled with EnableVariableCache.
- If also ValueChangeCallbackFunctions or FlagsChangeCallbackFunctions should be registered then see RegisterOnNodeModified.

par `HC` is the handle to the controller

`HNO` ist the handle to the NodeObject

`userpointer` is a pointer that is set by the user during registering the callback

`reserved` is reserved for later use

ret should return `0`

see 4.5.12.1 RegisterOnNodeModified, 4.5.2.6 EnableVariableCache

4.2.1.2. ValueChangeCallbackFunctionExt

fct

```
int (* ValueChangeCallbackFunctionExt(HController HC,
HNodeObject HNO, void *userpointer));
```

Functions of this type can be registered as callback functions for ValueChange-events (RegisterOnValueChangedExt).

par `HC` is the handle to the controller

HNO is the handle to the NodeObject

userpointer is a pointer that is set by the user during registering the callback

ret should return `0`

see 4.2.1.3 ValueChangeCallbackFunction,
4.5.12.6 RegisterOnValueChangedExt

4.2.1.3. ValueChangeCallbackFunction

fct

```
int (* ValueChangeCallbackFunction(HController HC,
HNodeObject HNO));
```

Functions of this type can be registered as callback functions for ValueChange-events (RegisterOnValueChanged).

par **HC** is the handle to the controller

HNO ist the handle to the NodeObject

ret should return `0`

see 4.2.1.2 ValueChangeCallbackFunctionExt,
4.5.12.10 RegisterOnValueChanged

4.2.1.4. FlagsChangeCallbackFunctionExt

fct

```
int (* FlagsChangeCallbackFunctionExt(HController HC,
HNodeObject HNO, void *userpointer));
```

Functions of this type can be registered as callback functions for FlagsChange-events (RegisterOnFlagsChangedExt).

TIP

Flags are normally only used for debugging.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`userpointer` is a pointer that is set by the user during registering the callback

ret should return `0`

see 4.2.1.5 FlagsChangeCallbackFunction,
4.5.12.13 RegisterOnFlagsChangedExt

4.2.1.5. FlagsChangeCallbackFunction

fct

```
int (* FlagsChangeCallbackFunction(HController HC,
HNodeObject HNO));
```

Functions of this type can be registered as callback functions for FlagsChange-events (RegisterOnFlagsChanged).

> **TIP**
>
> Flags are normally only used for debugging.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

ret should return `0`

see 4.2.1.4 FlagsChangeCallbackFunctionExt,
4.5.12.17 RegisterOnFlagsChanged

4.2.1.6. NameChangeCallbackFunctionExt

fct

```
int (* NameChangeCallbackFunctionExt(HController HC,
HNodeObject HNO, void *userpointer));
```

Functions of this type can be registered as callback functions for NameChange-events (RegisterOnNameChangedExt). The new name of the NodeObject can be obtained with GetNodeName.

> **TIP**
>
> In order to work correctly, also a ValueChangeCallbackFunction has to be registered for the same variable.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`userpointer` is a pointer that is set by the user during registering the callback

ret should return `0`

see 4.2.1.7 NameChangeCallbackFunction,
4.2.1.3 ValueChangeCallbackFunction,
4.5.12.48 RegisterOnNameChangedExt, 4.5.7.42 GetNodeName

4.2.1.7. NameChangeCallbackFunction

fct

```
int (* NameChangeCallbackFunction(HController HC,
HNodeObject HNO));
```

Functions of this type can be registered as callback functions for NameChange-events (RegisterOnNameChanged). The new name of the NodeObject can be obtained with GetNodeName.

> **TIP**
>
> In order to work correctly, also a ValueChangeCallbackFunction has to be registered for the same variable.

par **`HC`** handle to the controller

`HNO` handle to the NodeObject

ret should return `0`

see 4.2.1.6 NameChangeCallbackFunctionExt,
4.2.1.3 ValueChangeCallbackFunction,
4.5.12.53 RegisterOnNameChanged, 4.5.7.42 GetNodeName

4.2.1.8. NodeCreatedCallbackFunctionExt

fct

```
int (* NodeCreatedCallbackFunctionExt(HController HC, const
 char *varname, void *userpointer));
```

Functions of this type can be registered as callback functions for NodeCreated-events (RegisterOnNodeCreatedExt).

par **`HC`** is the handle to the controller

`varname` is the complete path to the variable

`userpointer` is a pointer that is set by the user during registering the callback

ret should return `0`

see 4.2.1.9 NodeCreatedCallbackFunction,
4.5.12.25 RegisterOnNodeCreatedExt

4.2.1.9. NodeCreatedCallbackFunction

fct

```
int (* NodeCreatedCallbackFunction(HController HC, const
char *varname));
```

Functions of this type can be registered as callback functions for NodeCreated-events (RegisterOnNodeCreated).

par **`HC`** is the handle to the controller

`varname` is the complete path to the variable

ret should return `0`

see 4.2.1.8 NodeCreatedCallbackFunctionExt,
4.5.12.27 RegisterOnNodeCreated

4.2.1.10. StartOfNodeCreatedRequestCallbackFunction

fct

```
int (* StartOfNodeCreatedRequestCallbackFunction(
HController HC, void *userpointer));
```

Functions of this type can be registered as callback functions to notify the user that a new set of NodeCreated-events is coming. This is useful when creating a set of new job nodes. Calling EndOfNodeCreatedRequestCallbackFunction guarantees that all needed variables for a `VAR:SET` are created.

par **`HC`** is the handle to the controller

`userpointer` is a pointer that is set by the user during registering the callback

ret should return `0`

see 4.2.1.11 EndOfNodeCreatedRequestCallbackFunction,
4.5.12.30 RegisterOnStartOfNodeCreatedRequest,
4.2.1.8 NodeCreatedCallbackFunctionExt,
4.5.12.27 RegisterOnNodeCreated

4.2.1.11. EndOfNodeCreatedRequestCallbackFunction

fct

```
int (* EndOfNodeCreatedRequestCallbackFunction(HController
HC, void *userpointer));
```

Functions of this type can be registered as callback functions to notify the user that a set of NodeCreated-events is finished. This is useful when creating a set of new job nodes. Calling EndOfNodeCreatedRequestCallbackFunction guarantees that all needed variables for a `VAR:SET` are created.

par **`HC`** is the handle to the controller

`userpointer` is a pointer that is set by the user during registering the callback

ret should return `0`

see 4.2.1.10 StartOfNodeCreatedRequestCallbackFunction,
4.5.12.32 RegisterOnEndOfNodeCreatedRequest,
4.2.1.8 NodeCreatedCallbackFunctionExt,
4.5.12.27 RegisterOnNodeCreated

4.2.1.12. NodeDeletedCallbackFunctionExt

fct

```
int (* NodeDeletedCallbackFunctionExt(HController HC,
HNodeObject HNO, void *userpointer));
```

Functions of this type can be registered as callback functions for NodeDeleted-events (RegisterOnNodeDeletedExt). If this function is called then the corresponding variable on the controller is already deleted. This function can be used to call DeleteNode on the given NodeObject.

TIP

In order to work correctly, also a ValueChangeCallbackFunction has to be registered for the same variable.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

`userpointer` is a pointer that is set by the user during registering the callback

ret should return `0`

see 4.2.1.16 NodeWillBeDeletedCallbackFunction,
4.2.1.13 NodeDeletedCallbackFunction,
4.2.1.3 ValueChangeCallbackFunction,
4.5.12.34 RegisterOnNodeDeletedExt 4.5.7.23 DeleteNode

4.2.1.13. NodeDeletedCallbackFunction

fct

```
int (* NodeDeletedCallbackFunction(HController HC,
HNodeObject HNO));
```

Functions of this type can be registered as callback functions for NodeDeleted-events (RegisterOnNodeDeleted). If this function is called then the corresponding variable on the controller is already deleted. This function can be used to call DeleteNode on the given NodeObject.

TIP

In order to work correctly, also a ValueChangeCallbackFunction has to be registered for the same variable.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

ret should return `0`

see 4.2.1.16 NodeWillBeDeletedCallbackFunction,
4.2.1.12 NodeDeletedCallbackFunctionExt,
4.2.1.3 ValueChangeCallbackFunction, 4.5.12.39 RegisterOnNodeDeleted
4.5.7.23 DeleteNode

4.2.1.14. StartOfNodeDeletedRequestCallbackFunction

fct

```
int (* StartOfNodeDeletedRequestCallbackFunction(
HController HC, void *userpointer));
```

Functions of this type can be registered as callback functions to notify the user that a new set of NodeDeleted-events is coming.

par `HC` is the handle to the controller

`userpointer` is a pointer that is set by the user during registering the callback

ret should return `0`

see 4.2.1.15 EndOfNodeDeletedRequestCallbackFunction,
4.5.12.42 RegisterOnStartOfNodeDeletedRequest,
4.2.1.12 NodeDeletedCallbackFunctionExt,
4.5.12.39 RegisterOnNodeDeleted

4.2.1.15. EndOfNodeDeletedRequestCallbackFunction

fct

```
int (* EndOfNodeDeletedRequestCallbackFunction(HController
HC, void *userpointer));
```

Functions of this type can be registered as callback functions to notify the user that a set of NodeDeleted-events is finished.

par `HC` is the handle to the controller

`userpointer` is a pointer that is set by the user during registering the callback

ret should return `0`

see 4.2.1.14 StartOfNodeDeletedRequestCallbackFunction,
4.5.12.44 RegisterOnEndOfNodeDeletedRequest,
4.2.1.12 NodeDeletedCallbackFunctionExt,
4.5.12.39 RegisterOnNodeDeleted

4.2.1.16. NodeWillBeDeletedCallbackFunction

fct

```
int (* NodeWillBeDeletedCallbackFunction(HController HC,
HNodeObject HNO, void *userpointer));
```

Functions of this type can be registered as callback functions for NodeWillBeDeleted-events (**??**). This function gets called before all the NodeDeleted-events are coming and telling you the root-node of the deleted variables.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

`userpointer` is a pointer that is set by the user during registering the callback

ret should return `0`

see 4.2.1.13 NodeDeletedCallbackFunction,
4.2.1.3 ValueChangeCallbackFunction,
4.5.12.34 RegisterOnNodeDeletedExt 4.5.7.23 DeleteNode

4.2.1.17. NodeMovedCallbackFunction

fct

```
int (* NodeMovedCallbackFunction(HController HC,
HNodeObject HNO, HNodeObject FormerParent, HNodeObject
NewParent, void *userpointer));
```

Functions of this type can be registered as callback functions for NodeMoved-events (RegisterOnNodeMoved). At the moment of the call to this function, the node already moved and is a child of NewParent. Please note, that it might be possible, that some of the handles have `ARG_INVALID_HANDLE_VALUE`.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

FormerParent is the parent node before the movement

NewParent is the parent node after the movement

userpointer is a pointer that is set by the user during registering the callback

ret should return `0`

see 4.2.1.17 NodeMovedCallbackFunction, 4.5.12.20 RegisterOnNodeMoved

4.2.1.18. NodeMovedCallbackFunctionExt

fct

```
int (* NodeMovedCallbackFunctionExt(HController HC,
HNodeObject HNO, HNodeObject FormerParent, HNodeObject
NewParent, int oldindex, int newindex, void *userpointer));
```

Functions of this type can be registered as callback functions for NodeMoved-events (RegisterOnNodeMoved). At the moment of the call to this function, the node already moved and is a child of NewParent. Please note, that it might be possible, that some of the handles have `ARG_INVALID_HANDLE_VALUE`.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

FormerParent is the parent node before the movement

NewParent is the parent node after the movement

oldindex is the index before the movement

newindex is the index after the movement

userpointer is a pointer that is set by the user during registering the callback

ret should return `0`

see 4.2.1.17 NodeMovedCallbackFunction, 4.5.12.20 RegisterOnNodeMoved

4.2.1.19. NodeStateChangeCallbackFunction

fct

```
int (* NodeStateChangeCallbackFunction(HController HC,
HNodeObject HNO, int newstate, int oldstate, int error,
void *userpointer));
```

Functions of this type can be registered as callback functions for StateChange-events (RegisterOnNodeStateChanged). HC is a handle of the involved Controller and HNO the handle of the NodeObject, newstate is the state, the NodeObject is now in, if error is not E_OK an error has occured in the statemachine. The application should be terminated. Userpointer is a pointer set by the user during registering the callback.

For supported NodeObject states see NodeObject states.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`newstate` is the state the variable is now in

`oldstate` is the state the variable was before

`error` if not `E_OK` an error has occured

`userpointer` is a pointer that is set by the user during registering the callback

ret should return `0`

see 4.5.12.56 RegisterOnNodeStateChanged, 4.3.1 NodeObject states

4.2.2. PLC changes

Table of contents

4.2.2.1. PLCChangeCallbackFunctionExt 35
4.2.2.2. PLCChangeCallbackFunction 36

4.2.2.1. PLCChangeCallbackFunctionExt

fct

```
int (* PLCChangeCallbackFunctionExt(HController HC,
unsigned int value, unsigned int reserved, void *
userpointer));
```

Functions of that type can be registered as a callback for PLCChanged-events (RegisterOnPLCChangedExt). PLC states provide the user with information about the internal state of the controller. This means, that these states can change unpredictably because of external signals. It is possible, that more than one of this states are active at the same time.

Listing 4.4: Checking if `PLC_JOB_READY` is set

```
if ( (value & (1 << PLC_JOB_READY)) == (1 <<
    PLC_JOB_READY)) )
{
  // PLC_JOB_READY
}
```

For supported PLC states see PLC states.

par **`HC`** is the handle to the controller

`value` is the PLC state

`reserved` is reserved for later use

`userpointer` is a pointer that is set by the user during registering the callback

ret should return `0`

see 4.3.6 PLC states, 4.2.2.2 PLCChangeCallbackFunction, 4.5.12.61 RegisterOnPLCChangedExt

4.2.2.2. PLCChangeCallbackFunction

fct

```
int (* PLCChangeCallbackFunction(HController HC, unsigned
int value, unsigned int reserved));
```

Functions of that type can be registered as a callback for PLCChanged-events (RegisterOnPLCChanged). PLC states give the user information about the internal state of the controller. This means, that these states can change unpredictably because of external signals. It is possible, that more than one of these states are active at the same time.

Listing 4.5: Checking if `PLC_JOB_READY` is set

```
if ( (value & (1 << PLC_JOB_READY)) == (1 <<
    PLC_JOB_READY)) )
{
  // PLC_JOB_READY
}
```

For supported PLC states see PLC states.

par **`HC`** is the handle to the controller

`value` is the PLC state

`reserved` is reserved for later use

ret should return `0`

see 4.3.6 PLC states, 4.2.2.1 PLCChangeCallbackFunctionExt, 4.5.12.61 RegisterOnPLCChangedExt

4.2.3. System messages

Table of contents

4.2.3.1. SystemMessageCallbackFunction 37
4.2.3.2. SysMsgCallbackFunctionExt 37
4.2.3.3. SysMsgCallbackFunction 38

4.2.3.1. SystemMessageCallbackFunction

fct

```
int (* SystemMessageCallbackFunction(HSysMsg HSM, void *
userpointer));
```

Functions of that type can be registered as a callback function for SysMsg-events (RegisterOnSysMsgExt).

par **`HSM`** is the handle to the SysMessage

`userpointer` is a pointer that is set by the user during registering the callback

ret should return `0`

see 4.5.12.71 RegisterOnSystemMessage, 4.5.13.2 GetSystemMessageXML

4.2.3.2. SysMsgCallbackFunctionExt

fct

```
int (* SysMsgCallbackFunctionExt(HController HC, HSysMsg
HSM, void *userpointer));
```

⚠ NOTICE

This function is deprecated.

- Use SystemMessageCallbackFunction instead.

Functions of that type can be registered as a callback function for SysMessage-events (RegisterOnSysMsgExt).

par **`HC`** is the handle to the controller

`HSM` is the handle to the SysMessage

`userpointer` is a pointer that is set by the user during registering the callback

ret should return `0`

see 4.2.3.3 SysMsgCallbackFunction, 4.5.12.66 RegisterOnSysMsgExt, 4.2.3.1 SystemMessageCallbackFunction

4.2.3.3. SysMsgCallbackFunction

fct

```
int (* SysMsgCallbackFunction(HController HC, HSysMsg HSM));
```

> ⚠ **NOTICE**
>
> This function is deprecated.
>
> - Use SystemMessageCallbackFunction instead.

Functions of that type can be registered as a callback function for SysMessage-events (RegisterOnSysMsg).

par **`HC`** is the handle to the controller

`HSM` is the handle to the SysMessage

ret should return `0`

see 4.2.3.2 SysMsgCallbackFunctionExt, 4.5.12.68 RegisterOnSysMsg, 4.2.3.1 SystemMessageCallbackFunction

4.2.4. Devices

Table of contents

4.2.4.1. DeviceCreatedCallbackFunction 39
4.2.4.2. DeviceDeletedCallbackFunction 39
4.2.4.3. DeviceActivatedCallbackFunction 39
4.2.4.4. DeviceDeactivatedCallbackFunction 40
4.2.4.5. DeviceDependencyAddedCallbackFunction 40
4.2.4.6. DeviceDependencyRemovedCallbackFunction 40
4.2.4.7. DeviceStateChangedCallbackFunction 40
4.2.4.8. DeviceErrorStateChangedCallbackFunction 40
4.2.4.9. DeviceParamStateChangedCallbackFunction 41
4.2.4.10. DevicePowerStateChangedCallbackFunction 41

4.2.4.1. DeviceCreatedCallbackFunction

fct

```
int (* DeviceCreatedCallbackFunction(HController HC,
HDevice HD, void *userpointer));
```

4.2.4.2. DeviceDeletedCallbackFunction

fct

```
int (* DeviceDeletedCallbackFunction(HController HC,
HDevice HD, void *userpointer));
```

4.2.4.3. DeviceActivatedCallbackFunction

fct

```
int (* DeviceActivatedCallbackFunction(HController HC,
HDevice HD, void *userpointer));
```

4.2.4.4. DeviceDeactivatedCallbackFunction

fct

```
int (* DeviceDeactivatedCallbackFunction(HController HC,
HDevice HD, void *userpointer));
```

4.2.4.5. DeviceDependencyAddedCallbackFunction

fct

```
int (* DeviceDependencyAddedCallbackFunction(HController HC,
 HDevice ParentDevice, HDevice SubDevice, const char *
depvar, void *userpointer));
```

4.2.4.6. DeviceDependencyRemovedCallbackFunction

fct

```
int (* DeviceDependencyRemovedCallbackFunction(HController
HC, HDevice ParentDevice, HDevice SubDevice, const char *
depvar, void *userpointer));
```

4.2.4.7. DeviceStateChangedCallbackFunction

fct

```
int (* DeviceStateChangedCallbackFunction(HController HC,
HDevice HD, ARG_DEVICE_ACTIVATION_STATE oldstate,
ARG_DEVICE_ACTIVATION_STATE newstate, void *userpointer););
```

4.2.4.8. DeviceErrorStateChangedCallbackFunction

fct

```
int (* DeviceErrorStateChangedCallbackFunction(HController
HC, HDevice HD, ARG_DEVICE_ERROR_STATE oldstate,
ARG_DEVICE_ERROR_STATE newstate, void *userpointer););
```

4.2.4.9. DeviceParamStateChangedCallbackFunction

fct

```
int (* DeviceParamStateChangedCallbackFunction(HController
HC, HDevice HD, ARG_DEVICE_PARAM_STATE oldstate,
ARG_DEVICE_PARAM_STATE newstate, void *userpointer););
```

4.2.4.10. DevicePowerStateChangedCallbackFunction

fct

```
int (* DevicePowerStateChangedCallbackFunction(HController
HC, HDevice HD, ARG_DEVICE_POWER_STATE oldstate,
ARG_DEVICE_POWER_STATE newstate, void *userpointer););
```

4.2.5. Error, RequestLog, Log, JobLines

Table of contents

4.2.5.1. ErrorCallbackFunction 42
4.2.5.2. RequestLogCallbackFunction 43
4.2.5.3. LogCallbackFunction 43
4.2.5.4. JobLinesCallbackFunction 44

4.2.5.1. ErrorCallbackFunction

fct

```
int (* ErrorCallbackFunction(int errorcode, const char *
description, HController HC));
```

Functions of that type can be registered as a callback function for Error-events (RegisterOnError).

TIP

Functions of that type should be registered immediately after a call to InitControllerLib.

par **`errorcode`** `ERROR_CONNECTION_LOST` occurs, if the controller lost connection, e.g. if the controller has been shutdown or reset

`description` (information about this error is unavailable)

`HC` Handle to the controller which lost the connection

– OR –

`errorcode` `ERROR_OCCUPIED` occurs, if the connection to a controller could not be established

`description` holds the IP address that occupies the controller

`HC` `ARG_INVALID_HANDLE_VALUE`

ret should return `0`

see 4.5.3.1 RegisterOnError, 4.5.1.1 InitControllerLib

4.2.5.2. RequestLogCallbackFunction

fct

```
int (* RequestLogCallbackFunction(HController HC, const
char *logentry, void *userpointer));
```

Functions of that type can be registered as a callback function for RequestLog-events (RegisterOnRequestLog).

par **`HC`** is the handle to the controller

`logentry` is the log entry as a single string

`userpointer` is a pointer that is set by the user during registering the callback

ret should return `0`

see 4.5.16.1 RegisterOnRequestLog

4.2.5.3. LogCallbackFunction

fct

```
int (* LogCallbackFunction(ARG_LOG_LEVEL level, const char
*datetime, const char *logentry, void *userpointer));
```

Functions of that type can be registered as a callback function for Log-events (RegisterOnLog). Valid log levels are:
`LOG_SUCCESS`, `LOG_INFO`, `LOG_WARNING`,
`LOG_ERROR`

par **`HC`** is the handle to the controller

`datetime` is the current time in format `yyyy-mm-dd hh:mm:ss`

`logentry` is the log entry as a single string

`userpointer` is a pointer that is set by the user during registering the callback

ret should return `0`

see 4.5.16.6 RegisterOnLog

4.2.5.4. JobLinesCallbackFunction

fct

```
int (* JobLinesCallbackFunction(HController HC,
ARG_LINEINFO *lineinfo));
```

Gets called when GetJobLines has lineinfo.

par **HC** is the handle to the controller

see 4.5.9.9 GetJobLines

4.3. Supported states

4.3.1. NodeObject states

- NODESTATE_UNDEFINED
- NODESTATE_CREATING
- NODESTATE_UNINITIALIZED
- NODESTATE_VALUEVALID
- NODESTATE_VALUEINVALID
- NODESTATE_CLIENTCHANGING
- NODESTATE_CLIENTVALIDATING
- NODESTATE_CONTROLLERVALIDATING
- NODESTATE_CONTROLLERCHANGING
- NODESTATE_VALUEACCEPTED
- NODESTATE_DELETING

4.3.2. Device states

These states are used during the activation of a device. Errors during activation can be seen in this states. If a device is in the ARG_DEV_ACTIVE-state, then errors in the device can be seen with the 4.3.3 Device error states.

- ARG_DEV_UNINITIALIZED
- ARG_DEV_INACTIVE
- ARG_DEV_ACTIVE
- ARG_DEV_ACTIVATING
- ARG_DEV_DEACTIVATING
- ARG_DEV_ACTIVATIONERROR
- ARG_DEV_DEACTIVATIONERROR
- ARG_DEV_UNDEFINED

4.3.3. Device error states

- ARG_DEV_ERR_UNINITIALIZED
- ARG_DEV_ERR_OK
- ARG_DEV_ERR_RECOVER
- ARG_DEV_ERR_RECOVERERROR
- ARG_DEV_ERR_FAILURE
- ARG_DEV_ERR_NOTREADY
- ARG_DEV_ERR_UNDEFINED

4.3.4. Device parameter states

- ARG_DEV_PARAM_UNINITIALIZED
- ARG_DEV_PARAM_IDLE
- ARG_DEV_PARAM_EVAL
- ARG_DEV_PARAM_REEVAL
- ARG_DEV_PARAM_PRESUSPEND
- ARG_DEV_PARAM_SUSPENDED
- ARG_DEV_PARAM_UNDEFINED

4.3.5. Device power states

- ARG_DEV_POWER_UNINITIALIZED
- ARG_DEV_POWER_CHECKREADYTOSTANDBY
- ARG_DEV_POWER_DIRECTOFF
- ARG_DEV_POWER_DOWN
- ARG_DEV_POWER_GOINGDOWNTOSTANDBY
- ARG_DEV_POWER_GOINGDOWNTOOFF
- ARG_DEV_POWER_GOINGOFFTODOWN
- ARG_DEV_POWER_GOINGREADYTOSTANDBY
- ARG_DEV_POWER_GOINGSTANDBYTODOWN
- ARG_DEV_POWER_GOINGSTANDBYTOREADY
- ARG_DEV_POWER_OFF
- ARG_DEV_POWER_READY
- ARG_DEV_POWER_STANDBY
- ARG_DEV_POWER_UNDEFINED

4.3.6. PLC states

Depending on the controller firmware in use the supported PLC states may differ. This subset works with firmware versions equal to or greater than 2.4.0. For more information, please consult the documentation of the firmware in use [2].

TIP

See 4.2.2.2 PLCChangeCallbackFunction or 4.2.2.1 PLCChangeCallbackFunctionExt for a code example of testing for PLC states.

PLC_JOB_READY If this bit is set then the job is ready for execution. When 4.5.4.2 JobStart is executed then this state will go away and `PLC_JOB_ACTIVE` will be set instead

PLC_JOB_ACTIVE If this bit is set then the job is running on the controller and can be aborted with 4.5.4.4 JobAbort

PLC_DEVICES_FAILURE If this bit is set then one or more devices, that are controlled by the SAS-device, are in failure state

PLC_DEVICES_READY If this bit is set then all devices, that are controlled by the SAS-Device, are in ready state

4.4. Constants

The following constants are return values. Find the explanation for the respective constant in the functions where the constant occurs.

4.4.1. Error codes

- ARG_INVALID_HANDLE_VALUE
- E_OK
- E_FAILURE

- E_TIMEOUT
- E_NOSPACE
- E_EXIST
- E_NOEXIST
- E_UNIMPL

4.4.2. Variable types

- VT_INV
- VT_STR
- VT_SEL
- VT_BOOLEAN
- VT_BIN
- VT_I32
- VT_I64
- VT_R32
- VT_R64

4.4.3. PLC states

Find supported PLC states in 4.3.6 PLC states. For more information, please consult the documentation of the firmware in use; see [2].

- PLC_DEVICES_AWAKE
- PLC_DEVICES_FAILURE
- PLC_DEVICES_READY
- PLC_DEVICES_SETUP

- PLC_JOB_READY
- PLC_JOB_ACTIVE
- PLC_JOB_COMPLETED
- PLC_JOB_FAILED
- PLC_SYS_SAFE
- PLC_JOB_PAUSED
- PLC_JOB_PILOTED
- PLC_JOB_PRELOADED
- PLC_JOB_STOPPING
- PLC_READY_FOR_POWER_OFF
- PLC_SYSTEM_READY
- PLC_ATTENTION
- PLC_SYS_SAFE_REQUEST
- PLC_JOB_PRELOAD

4.5. Functions

4.5.1. General

Table of contents

4.5.1.1. InitControllerLib 50
4.5.1.2. DeinitControllerLib 51
4.5.1.3. IsInitialized 51
4.5.1.4. GetControllerLibVersionString 51
4.5.1.5. CLFree 52
4.5.1.6. DisableFloatingPointToStringDot 52
4.5.1.7. SetApplicationName 52
4.5.1.8. SetApplicationVersion 53
4.5.1.9. GetApplicationName 53
4.5.1.10. SetFullFeatured 54
4.5.1.11. IsFullFeatured 54
4.5.1.12. SetAttributes 54
4.5.1.13. SetInternalAttributes 55
4.5.1.14. EnableWatchdog 55

4.5.1.1. InitControllerLib

fct

```
int InitControllerLib(void);
```

Initializes the library.

This function has to be called before any other call to the library. Otherwise any calls to the library will not work.

Directly after this call we recommend to call RegisterOnError.

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.1.2 DeinitControllerLib, 4.5.3.1 RegisterOnError,
4.5.1.4 GetControllerLibVersionString, 4.5.1.3 IsInitialized

4.5.1.2. DeinitControllerLib

fct

```
int DeinitControllerLib(void);
```

Deinitializes the library.

This function has to be called before the client application exits.

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.1.1 InitControllerLib, 4.5.1.3 IsInitialized

4.5.1.3. IsInitialized

fct

```
int IsInitialized(void);
```

Returns, whether the library is already initialized or not.

ret `1` if the library is initialized

`0` if the library is not initialized

see 4.5.1.1 InitControllerLib, 4.5.1.2 DeinitControllerLib

4.5.1.4. GetControllerLibVersionString

fct

```
const char* GetControllerLibVersionString(void);
```

Gets the version of the library as a string.

ret string composed of the
⟨`library version`⟩.⟨`major version`⟩.⟨`minor version`⟩; e.g. “`1.1.6`”

see 4.5.1.1 InitControllerLib

4.5.1.5. CLFree

fct

```
void CLFree(void *pointer);
```

Frees memory allocated by the ControllerLib as the function `free()` cannot free this memory due to ownership issues on the Windows® operating system.

see 4.5.5.2 GetPenPath, 4.5.5.4 GetFontPath

4.5.1.6. DisableFloatingPointToStringDot

fct

```
void DisableFloatingPointToStringDot();
```

The firmware relies on a period (dot) as decimal separator when floating point numbers are passed as strings. When calling the functions GetNodeValueString, SetNodeValueString, GetNodeInfo or GetNodeInfoExt these will automatically correct accidental commas into periods (dots) in the string representations of floating point numbers. Calling function DisableFloatingPointToStringDot disables this behavior and is not recommended.

see 4.5.7.87 SetNodeValueString, 4.5.7.82 GetNodeValueString,
4.5.7.3 GetNodeInfo, 4.5.7.4 GetNodeInfoExt

4.5.1.7. SetApplicationName

fct

```
void SetApplicationName(const char *applicationname);
```

This function tells the controller's firmware the name of the client application.

As the string is initially empty the function should be called before calling function DetectRemoteController.

par **`applicationname`** is the name of the client application

see 4.5.1.8 SetApplicationVersion, 4.5.1.9 GetApplicationName, 4.5.1.10 SetFullFeatured, 4.5.2.2 DetectRemoteController

4.5.1.8. SetApplicationVersion

fct

```
void SetApplicationVersion(int major, int minor, int
modification, int buildnr, char *versiontype);
```

This function tells the controller's firmware the version of the client application.

The function should be called before calling function DetectRemoteController.

par **`major`** major nr of the application

`minor` minor nr of the application

`modification` modification nrof the application

`buildnr` buildnr of the application

`versiontype` versiontype version of the application (e.g. debug, TP, Release...)

see 4.5.1.7 SetApplicationName, 4.5.1.10 SetFullFeatured, 4.5.2.2 DetectRemoteController

4.5.1.9. GetApplicationName

fct

```
const char* GetApplicationName(void);
```

Gets the name of the application that was set by using SetApplicationName.

ret name of the application

see 4.5.1.7 SetApplicationName, 4.5.1.11 IsFullFeatured

4.5.1.10. SetFullFeatured

fct

```
void SetFullFeatured(int flag);
```

Sets, whether the client application is *fully featured* or not. Fully featured means, that the controller can ask the application for additional data, e.g. pens, bitmaps. By default the application is assumed as fully featured.

This function must be called before calling function DetectRemoteController.

par **`flag`** sets the client application as fully featured (`1`) or not fully featured (`0`)

see 4.5.1.11 IsFullFeatured, 4.5.1.7 SetApplicationName, 4.5.2.2 DetectRemoteController

4.5.1.11. IsFullFeatured

fct

```
int IsFullFeatured(void);
```

Returns, whether the client application is fully featured. This means, that the controller can ask the application for additional data, e.g. pens, bitmaps. By default the application is assumed as fully featured.

ret **`1`** if the application is fully featured

`0` if the application is not fully featured

see 4.5.1.10 SetFullFeatured, 4.5.1.7 SetApplicationName

4.5.1.12. SetAttributes

fct

```
void SetAttributes(void);
```

This function tells the ControllerLib to use attributes.

This function has to be called before calling DetectRemoteController.

see 4.5.1.13 SetInternalAttributes, 4.5.2.2 DetectRemoteController

4.5.1.13. SetInternalAttributes

fct
```
void SetInternalAttributes(void);
```

This function tells the ControllerLib to use firmware-internal attributes. The function is only used for debugging purposes.

This function has to be called before calling function DetectRemoteController.

see 4.5.1.12 SetAttributes, 4.5.2.2 DetectRemoteController

4.5.1.14. EnableWatchdog

fct
```
void EnableWatchdog(void);
```

Sets, whether the client application will have the watchdog running or not. By default, the watchdog is disabled. The watchdog will close the connection, if the firmware seems to hang or the controller was turned off.

This function must be called before calling function DetectRemoteController.

par `flag` Enables the watchdog

see 4.5.1.11 IsFullFeatured, 4.5.1.7 SetApplicationName,
4.5.2.2 DetectRemoteController

4.5.2. Controller handling

Table of contents

4.5.2.1. ProbeRemoteController . . . 56
4.5.2.2. DetectRemoteController . . . 57
4.5.2.3. GetFeatureList . . . 57
4.5.2.4. SupportsXMLJobFormat . . . 58
4.5.2.5. DestroyFeatureList . . . 59
4.5.2.6. EnableVariableCache . . . 59
4.5.2.7. ControllerCount . . . 60
4.5.2.8. GetCorrectionGrid . . . 60
4.5.2.9. FreeCorrectionGridData . . . 61
4.5.2.10. DisconnectController . . . 61
4.5.2.11. GetUniqueHandle . . . 61
4.5.2.12. GetControllerRCCFile . . . 62
4.5.2.13. GetControllerQCHFile . . . 63
4.5.2.14. GetControllerQHCFile . . . 64
4.5.2.15. FreeRCCFile . . . 65
4.5.2.16. FreeQCHFile . . . 65
4.5.2.17. FreeQHCFile . . . 65

4.5.2.1. ProbeRemoteController

fct

```
int ProbeRemoteController(const char *host, short port);
```

Probes, whether a controller can be accessed at the given IP-address. The probing can last up to 5 seconds.

par **`host`** is the host name or IP-address of the remote controller, e.g. “`ascstack`”, “`192.168.1.42`”

`port` is the port of the remote controller, usually “`1610`”

ret **1** if a controller has been found

0 if a controller has not been found

see 4.5.2.2 DetectRemoteController

4.5.2.2. DetectRemoteController

fct

```
HController DetectRemoteController(const char *host, short
port);
```

Detects, whether a controller can be accessed at the given IP-address and, if this is true, establishes a connection and returns a handle.

par **host** is the host name or IP-address of the remote controller, e.g. “`ascstack`”, “`192.168.1.42`”

port is the port of the remote controller, usually “`1610`”

ret handle to the controller

ARG_INVALID_HANDLE_VALUE if a controller has not been found

see 4.5.2.1 ProbeRemoteController, 4.5.2.10 DisconnectController, 4.5.2.6 EnableVariableCache

4.5.2.3. GetFeatureList

fct

```
ARG_FEATURELIST* GetFeatureList(HController HC);
```

This function returns the features supported by the InScript firmware. This function always returns a valid pointer which never is `NULL`. If no features are supported by the InScript firmware then the feature count is `0`. The returned value has to be destroyed by using function DestroyFeatureList.

Listing 4.6: GetFeatureList example

```
ARG_FEATURELIST *list = GetFeatureList(HC);
for (int i=0; i<list->featurecount; ++i) {
  printf("Feature: %s  (Version: %i, Flags: %x)\n",
              list->feature[i]->name,
              list->feature[i]->version,
              list->feature[i]->flags
        );
}
DestroyFeatureList(list);
```

par **HC** is the handle to the controller

ret ARG_FeatureList for the controller

see 4.5.2.5 DestroyFeatureList

4.5.2.4. SupportsXMLJobFormat

fct

```
int SupportsXMLJobFormat(HController HC, int *version, int
*flags);
```

Returns, whether the firmware on the given controller supports jobs in the XML-format or not.

Listing 4.7: SupportsXMLJobFormat example

```
int version, flags;
if ( SupportsXMLJobFormat(HC, &version, &flags) == E_OK)
    {
  if ( version == 1 ) {
    // Load job in XML-format version 1 or higher
  }
} else {
  // Load job in "oldstyle"-format
}
```

par **HC** is the handle to the controller

version holds the highest supported version if the function returns E_OK

flags holds additional information if the function returns E_OK

ret **E_OK** if the firmware supports jobs in XML-format

E_UNAVAIL if the firmware does not support jobs in XML-format

E_FAILURE on error

see 4.5.2.5 DestroyFeatureList

4.5.2.5. DestroyFeatureList

fct

```
void DestroyFeatureList(ARG_FEATURELIST *list);
```

Releases the memory obtained by the call to GetFeatureList.

par **list** is a pointer to an ARG_FEATURELIST structure

see 4.5.2.3 GetFeatureList

4.5.2.6. EnableVariableCache

fct

```
int EnableVariableCache(HController HC);
```

Mirrors all variables on the controller to the ARG_ControllerLib. New variables will be added automatically, deleted variables will be deleted in the ARG_ControllerLib as well. Registered callbacks will also work. It is also possible to traverse through the tree with GetParentNode and GetSubnodes.

This method is preferred if many variables are used by the client application. It is always advisable to enable the variable cache to gain performance.

This function should be called immediately after DetectRemoteController.

> **TIP**
>
> See section 5.4 for an example of how to use the VariableCache.

par `HC` is the handle to the controller

ret `E_OK` on success

`E_EXIST` if the variable cache is already enabled

`E_FAILURE` on failure

see 4.5.2.2 DetectRemoteController, 4.5.7.15 GetParentNode, 4.5.7.2 GetNodeFromCache, 4.5.7.16 GetSubnodesCount, 4.5.8.6 GetSubnodes

4.5.2.7. ControllerCount

fct

```
int ControllerCount(void);
```

Gets the number of connected controllers.

ret number of connected controllers

see 4.5.2.2 DetectRemoteController

4.5.2.8. GetCorrectionGrid

fct

```
int GetCorrectionGrid(HController HC, ARG_GRID_DATA *data);
```

Please note, that this function may take several minutes to return.

ret `E_OK` on success

`E_NOMEM` if the data is `NULL`

`E_UNIMP` if the firmware does not support this feature

`E_FAILURE` on failure

4.5.2.9. FreeCorrectionGridData

fct
```
int FreeCorrectionGridData(ARG_GRID_DATA *data);
```

Frees the used internal correction data.

ret `E_OK` on success

`E_FAILURE` on failure

4.5.2.10. DisconnectController

fct
```
void DisconnectController(HController HC);
```

Disconnects controller HC. If the VariableCache was enabled with function EnableVariableCache no calls to DeleteNode are necessary.

par `HC` is the handle to the controller

see 4.5.2.2 DetectRemoteController, 4.5.2.6 EnableVariableCache, 4.5.7.23 DeleteNode

4.5.2.11. GetUniqueHandle

fct
```
DWORD GetUniqueHandle();
```

Gets an unique handle from the ControllerLib which will not be used internally.

ret unique handle

4.5.2.12. GetControllerRCCFile

fct

```
int GetControllerRCCFile(HController HC, void **data, int *
bytesize);
```

Gets the Qt®-RCC file from the controller.

Listing 4.8: GetControllerRCCFile example

```
int bytesize;
void *data;
if ( GetControllerRCCFile(HC, &data, &bytesize) == E_OK
   ) {
  printf("Read %d bytes\n");
  hexdump(data,bytesize);
  FreeRCCFile(data);
}
```

TIP

- This method is only needed for Qt®-applications which show integrated dialogs.
- The caller of this method is responsible for freeing the allocated memory.

par **HC** is the handle to the controller

data holds the binary data if the function returns `E_OK`

bytesize holds the binary data size if the function returns `E_OK`

ret **E_OK** on success

E_UNIMP if the firmware does not implement this feature

E_FAILURE on failure

see 4.5.2.15 FreeRCCFile

4.5.2.13. GetControllerQCHFile

fct

```
int GetControllerQCHFile(HController HC, void **data, int *
bytesize);
```

Gets the Qt®-help file from the controller.

Listing 4.9: GetControllerQCHFile example

```
int bytesize;
void *data;
if ( GetControllerQCHFile(HC, &data, &bytesize) == E_OK
   ) {
  printf("Read %d bytes\n");
  hexdump(data,bytesize);
  FreeQCHFile(data);
}
```

TIP

The caller of this method is responsible for freeing the allocated memory.

par **`HC`** is the handle to the controller

`data` holds the binary data if the function returns `E_OK`

`bytesize` holds the binary data size if the function returns `E_OK`

ret **`E_OK`** on success

`E_UNIMP` if the firmware does not implement this feature

`E_FAILURE` on failure

see 4.5.2.16 FreeQCHFile

4.5.2.14. GetControllerQHCFile

fct

```
int GetControllerQHCFile(HController HC, void **data, int *
bytesize);
```

Gets the Qt®-help file collection from the controller.

Listing 4.10: GetControllerQHCFile example

```
int bytesize;
void *data;
if ( GetControllerQHCFile(HC, &data, &bytesize) == E_OK
   ) {
  printf(”Read %d bytes\n”);
  hexdump(data,bytesize);
  FreeQHCFile(data);
}
```

TIP

The caller of this method is responsible for freeing the allocated memory.

par **HC** is the handle to the controller

data holds the binary data if the function returns `E_OK`

bytesize holds the binary data size if the function returns `E_OK`

ret **E_OK** on success

E_UNIMP if the firmware does not implement this feature

E_FAILURE on failure

see 4.5.2.17 FreeQHCFile

4.5.2.15. FreeRCCFile

fct

```
void FreeRCCFile(void *data);
```

Frees the data that was allocated by GetControllerRCCFile.

See GetControllerRCCFile for an example.

par **`data`** holds the data obtained by GetControllerRCCFile

see 4.5.2.12 GetControllerRCCFile

4.5.2.16. FreeQCHFile

fct

```
void FreeQCHFile(void *data);
```

Frees the data that was allocated by GetControllerQCHFile.

See GetControllerQCHFile for an example.

par **`data`** holds the data obtained by GetControllerQCHFile

see 4.5.2.13 GetControllerQCHFile

4.5.2.17. FreeQHCFile

fct

```
void FreeQHCFile(void *data);
```

Frees the data that was allocated by GetControllerQHCFile.

See GetControllerQHCFile for an example.

par **`data`** holds the data obtained by GetControllerQHCFile.

see 4.5.2.14 GetControllerQHCFile

4.5.3. Error handling

Table of contents

4.5.3.1. RegisterOnError 66
4.5.3.2. UnregisterOnError 66
4.5.3.3. UnregisterOnErrorSingle 67

4.5.3.1. RegisterOnError

fct

```
int RegisterOnError(ErrorCallbackFunction callback);
```

Registers a callback for error-events; see ErrorCallbackFunction.

TIP

It is possible to have more than one callback for each variable.

par `callback` is the callback function

ret `E_OK` on success

`E_EXIST` if a callback already exists

`E_FAILURE` on failure

see 4.5.3.2 UnregisterOnError, 4.5.3.3 UnregisterOnErrorSingle, 4.5.1.1 InitControllerLib, 4.2.5.1 ErrorCallbackFunction

4.5.3.2. UnregisterOnError

fct

```
int UnregisterOnError();
```

Unregisters all callbacks for error-events before DeinitControllerLib.

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.3.1 RegisterOnError, 4.5.3.3 UnregisterOnErrorSingle,
4.5.1.2 DeinitControllerLib

4.5.3.3. UnregisterOnErrorSingle

fct

```
int UnregisterOnErrorSingle(ErrorCallbackFunction callback);
```

Unregisters a callback for error-events before DeinitControllerLib.

par **`callback`** is the callback function

ret **`E_OK`** on success

`E_NOEXIST` if the callback was not registered

`E_FAILURE` on failure

see 4.5.3.1 RegisterOnError, 4.5.3.2 UnregisterOnError,
4.5.1.2 DeinitControllerLib

4.5.4. Requests

Table of contents

4.5.4.1. GetPLCState 68
4.5.4.2. JobStart 69
4.5.4.3. JobPilot 69
4.5.4.4. JobAbort 70
4.5.4.5. JobStop 70
4.5.4.6. ExecuteScript 71
4.5.4.7. WriteToNVRAM 72

4.5.4.1. GetPLCState

fct

```
int GetPLCState(HController HC, unsigned int *value,
unsigned int *reserved);
```

Gets the current PLC state of the controller. Note that you can register a callback function to get notified if the state changes. This call gets the current value. See PLC states for supported PLC states.

> **TIP**
>
> For performance reasons you should use RegisterOnPLCChanged instead of this function.

par **`HC`** is the handle to the controller

`value` holds the PLC-value after returning with `E_OK`

`reserved` is reserved for later use

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.12.63 RegisterOnPLCChanged, 4.3.6 PLC states

4.5.4.2. JobStart

fct

```
int JobStart(HController HC);
```

Sends a JobStart to the controller. If a job is selected and the controller is ready for job execution then the controller will start marking. This call returns immediately.

> **TIP**
>
> If this function returns successfully then this does not mean that the job execution has started. Job execution should be watched by using the PLC states.

par **`HC`** is the handle to the controller

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.6.1 LoadJob, 4.5.7.17 SelectNode, 4.5.4.3 JobPilot, 4.5.4.4 JobAbort, 4.5.4.5 JobStop, 4.5.12.63 RegisterOnPLCChanged

4.5.4.3. JobPilot

fct

```
int JobPilot(HController HC);
```

Sends a JobPilot (TeachInStart) to the controller.

> **TIP**
>
> If this function returns successfully then this does not mean that the job execution has started. Job execution should be watched by using the PLC states.

par **`HC`** is the handle to the controller

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.4.2 JobStart, 4.5.4.4 JobAbort, 4.5.4.5 JobStop,
4.5.12.63 RegisterOnPLCChanged

4.5.4.4. JobAbort

fct

```
int JobAbort(HController HC);
```

Sends a JobAbort to the controller. If a job is marking then the controller will abort the job execution. This call returns immediately.

> **TIP**
>
> If this function returns successfully then this does not mean that the job execution has been aborted. Job execution should be watched by using the PLC states.

par `HC` is the handle to the controller

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.4.2 JobStart, 4.5.4.3 JobPilot, 4.5.4.5 JobStop,
4.5.12.63 RegisterOnPLCChanged

4.5.4.5. JobStop

fct

```
int JobStop(HController HC);
```

Send a JobStop to the controller. If a job is marking then the controller will stop the job execution. This call returns immediately.

TIP

- JobStop only stops the job at a Stop-node in a job. If marking should be aborted immediately then use JobAbort instead.
- When this function returns successfully, this does not mean, that the job execution has been stopped. Job execution should be watched using the PLC states.

par `HC` is the handle to the controller

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.4.2 JobStart, 4.5.4.3 JobPilot, 4.5.4.4 JobAbort, 4.5.12.63 RegisterOnPLCChanged

4.5.4.6. ExecuteScript

fct

```
int ExecuteScript(HController HC, const char *script);
```

Executes the script on the controller. The script is executed immediately.

TIP

Please refer to the InScript manual [3] for a documentation on the scripting language.

Listing 4.11: ExecuteScript example

```
const char *script = "if node_exists("usr.var.x")
   delete_node("usr.var.x");"
ExecuteScript(HC, script);
```

par `HC` is the handle to the controller

`script` is the script that shall be executed

ret `E_OK` on success

`E_FAILURE` on failure

4.5.4.7. WriteToNVRAM

fct

```
int WriteToNVRAM(HController HC);
```

Saves the device configuration and the default pen to the controller. If the controller is restarted again then this data will be loaded.

par **HC** is the handle to the controller

ret **E_OK** on success

E_FAILURE on failure

4.5.5. Paths

Table of contents

4.5.5.1. SetPenPath 73
4.5.5.2. GetPenPath 73
4.5.5.3. SetFontPath 74
4.5.5.4. GetFontPath 74
4.5.5.5. GetAllAvailableFontnames 75
4.5.5.6. DestroyFontInfo 75

4.5.5.1. SetPenPath

fct

```
int SetPenPath(HController HC, const char *path);
```

Sets the path where pens are searched. See Loading a job for the search strategy.

par **`HC`** is the handle to the controller

`path` is the path were pens can be found

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.5.2 GetPenPath, 4.5.5.3 SetFontPath, 5.1.2 Loading a job

4.5.5.2. GetPenPath

fct

```
char* GetPenPath(HController HC);
```

Returns the path where pens are searched. See Loading a job for the search strategy.

TIP

The user has to free the allocated memory using CLFree.

Listing 4.12: GetPenPath example

```
char *path = GetPenPath(HC);
printf("Penpath: %s", path);
CLFree(path);
```

par `HC` is the handle to the controller

ret path were pens can be found

see 4.5.5.1 SetPenPath, 4.5.5.4 GetFontPath, 4.5.1.5 CLFree, 5.1.2 Loading a job

4.5.5.3. SetFontPath

fct

```
int SetFontPath(HController HC, const char *path);
```

Sets the path where TTF- and FDT-fonts are searched. See Loading a job for the search strategy.

par `HC` is the handle to the controller

`path` is the path were TTF- and FDT-fonts can be found

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.5.1 SetPenPath, 4.5.5.4 GetFontPath, 5.1.2 Loading a job

4.5.5.4. GetFontPath

fct

```
char* GetFontPath(HController HC);
```

Returns the path where TTF- and FDT-fonts are searched. See Loading a job for the search strategy.

TIP

The user has to free the allocated memory using CLFree.

Listing 4.13: GetFontPath example

```
const char *path = GetPenPath(HC);
printf("Penpath: %s", path);
CLFree(path);
```

par `HC` is the handle to the controller

ret the path were TTF- and FDT-fonts can be found

see 4.5.5.3 SetFontPath, 4.5.5.2 GetPenPath, 4.5.1.5 CLFree, 5.1.2 Loading a job

4.5.5.5. GetAllAvailableFontnames

fct

```
ARG_FONTINFO* GetAllAvailableFontnames(HController HC);
```

Returns a list with all font names which are available for the controller. This includes the fonts on the controller and all TTF- and FDT-fonts that were found in the `FontPath`.

par `HC` is the handle to the controller

ret an ARG_FONTINFO-structure with all information about font

see 4.5.5.6 DestroyFontInfo, 4.5.5.3 SetFontPath

4.5.5.6. DestroyFontInfo

fct

```
void DestroyFontInfo(ARG_FONTINFO *info);
```

Frees the list returned by GetAllAvailableFontnames.

par `info` is the structure returned by GetAllAvailableFontnames

see 4.5.5.5 GetAllAvailableFontnames

4.5.6. Files

Table of contents

4.5.6.1.	LoadJob	76
4.5.6.2.	LoadJobExt	78
4.5.6.3.	SaveJob	79
4.5.6.4.	SaveJobXML	80
4.5.6.5.	SaveTree	81
4.5.6.6.	SaveTreeXML	81
4.5.6.7.	SaveTreeAsString	82
4.5.6.8.	SaveTreeAsXMLString	83
4.5.6.9.	FreeSaveTreeString	84
4.5.6.10.	LoadTree	85
4.5.6.11.	LoadTreeFromString	85
4.5.6.12.	LoadTreeFromStringExt	86
4.5.6.13.	SavePen	87
4.5.6.14.	LoadPen	87
4.5.6.15.	LoadFont	88
4.5.6.16.	LoadDataXMLFromFile	89
4.5.6.17.	LoadDataXML	90

4.5.6.1. LoadJob

fct

```
int LoadJob(HController HC, const char *filename, int
clearfirst, int select);
```

Loads a job file to the controller. It is possible to load jobs that were saved by SaveJob or SaveJobXML.

> **TIP**
>
> **Pens** Please note, that extensions of pen files have to be lowercase (`.pen`). See Loading a job for details on where the pens and fonts are searched.
>
> **Fonts** Only TTF- and FDT-fonts are supported by the library. Please note, that the extension of the font files have to be lowercase (`.ttf` or `.fdt`). See Loading a job for details on where the pens and fonts are searched.
>
> **Bitmaps** Upload and streaming of a big set of bitmap formats is supported. The file name of a bitmap is the path as seen from the client application. Note that the bitmap is not searched in the job directory.

par **`HC`** is the handle to the controller

`filename` is the file name of the job. The path must be accessible from the client application

`clearfirst` `1`: clears any other job from the controller before uploading this job; `0`: leaves the jobs on the controller

`select` `1`: selects this job for execution after it has been loaded; `0` does not select this job for execution

ret **`E_OK`** on success

`E_FAILURE` on failure

`E_TIMEOUT` if the communication with the firmware times out

`E_NOSPACE` if there is not enough memory for the job

see 5.1.2 Loading a job, 5.5.5 Loading a job (Example), 4.5.6.3 SaveJob, 4.5.6.4 SaveJobXML, 4.5.9.4 GetJobNames, 4.5.9.5 JobClearAll, 4.5.7.17 SelectNode, 4.5.4.2 JobStart, 4.5.5.1 SetPenPath, 4.5.5.3 SetFontPath

4.5.6.2. LoadJobExt

fct

```
int LoadJobExt(HController HC, const char *filename, int
clearfirst, int select, char *retname, int retnamelen);
```

Loads a job file to the controller. It is possible to load jobs that were saved by SaveJob or SaveJobXML.

> **TIP**
>
> **Pens** Please note, that extensions of pen files have to be lowercase (*.pen*). See Loading a job for details on where the pens and fonts are searched.
>
> **Fonts** Only TTF- and FDT-fonts are supported by the library. Please note, that the extension of the font files have to be lowercase (`.ttf` or `.fdt`). See Loading a job for details on where the pens and fonts are searched.
>
> **Bitmaps** Upload and streaming of a big set of bitmap formats is supported. The file name of a bitmap is the path as seen from the client application. Note that the bitmap is not searched in the job directory.

par **`HC`** is the handle to the controller

`filename` is the file name of the job. The path must be accessible from the client application

`clearfirst` `1`: clears any other job from the controller before uploading this job; `0`: leaves the jobs on the controller

`select` `1`: selects this job for execution after it has been loaded; `0` does not select this job for execution

`retname` is the file name of the job that is returned by the firmware. The path must be accessible from the client application

`retnamelen` is the buffer lenght for the job name. The length is usually 32 bytes

ret `E_OK` on success

`E_FAILURE` on failure

`E_TIMEOUT` if the communication with the firmware times out

`E_NOSPACE` if there is not enough memory for the job

see 5.1.2 Loading a job, 5.5.5 Loading a job (Example), 4.5.6.3 SaveJob, 4.5.6.4 SaveJobXML, 4.5.9.4 GetJobNames, 4.5.9.5 JobClearAll, 4.5.7.17 SelectNode, 4.5.4.2 JobStart, 4.5.5.1 SetPenPath, 4.5.5.3 SetFontPath

4.5.6.3. SaveJob

fct

```
int SaveJob(HController HC, const char *jobname, const char
 *filename);
```

Saves a job to the file system.

par `HC` is the handle to the controller

`jobname` is the name of the job, e.g. `usr.job.MyJob` or simply `MyJob`

`filename` is the file name of the job. The path must be accessible from the client application

ret `E_OK` on success

`E_FAILURE` on failure

`E_TIMEOUT` if the communication with the firmware times out

see 4.5.6.1 LoadJob, 4.5.6.4 SaveJobXML, 4.5.6.5 SaveTree

4.5.6.4. SaveJobXML

fct

```
int SaveJobXML(HController HC, const char *jobname, const
char *filename, int includeDependencies, int resetModified);
```

Saves a job in the XML format on the controller.

> **TIP**
>
> Not every firmware supports this call. If the firmware supports this feature then the feature list (GetFeatureList) contains the feature *JobXMLFileFormat*.

par **`HC`** is the handle to the controller

`jobname` is the name of the job, e.g. `usr.job.MyJob` or simply `MyJob`

`filename` is the file name of the job. The path must be accessible from the client application

`includeDependencies` 1: if the job file shall include dependencies like pens and macros; `0`: otherwise

`resetModified` 1: if the job shall be set as *not modified* on the controller. This should be the default; `0`: otherwise

ret **`E_OK`** on success

`E_FAILURE` on failure

`E_UNIMP` if the firmware does not implement this feature

`E_TIMEOUT` if the communication with the firmware times out

see 4.5.6.1 LoadJob, 4.5.6.5 SaveTree 4.5.2.3 GetFeatureList

4.5.6.5. SaveTree

fct

```
int SaveTree(HController HC, HNodeObject HNO, const char *
filename);
```

Saves a subtree to the file system.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject from where to save the file

filename is the file name of the job. The path must be accessible from the client application

ret **E_OK** on success

E_FAILURE on failure

E_TIMEOUT if the communication with the firmware times out

see 4.5.6.10 LoadTree, 4.5.6.7 SaveTreeAsString, 4.5.6.8 SaveTreeAsXMLString, 4.5.6.3 SaveJob

4.5.6.6. SaveTreeXML

fct

```
int SaveTreeXML(HController HC, HNodeObject HNO, const char
 *filename);
```

Saves a subtree to the file system.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject from where to save the file

filename is the file name of the job. The path must be accessible from the client application

ret **E_OK** on success

E_FAILURE on failure

E_TIMEOUT if the communication with the firmware times out

see 4.5.6.10 LoadTree, 4.5.6.7 SaveTreeAsString, 4.5.6.8 SaveTreeAsXMLString, 4.5.6.3 SaveJob

4.5.6.7. SaveTreeAsString

fct

```
char* SaveTreeAsString(HController HC, HNodeObject HNO);
```

Gets a string representation of a subtree.

Listing 4.14: SaveTreeAsString example

```
HNodeObject HNOSrc = GetNode(HC, "usr.job.Job.Transform1
   ");
if ( HNOSrc != ARG_INVALID_HANDLE_VALUE ) {
  char *tmp = SaveTreeAsString(HC, HNOSrc);
  HNodeObject HNODst = GetNode(HC, "usr.job.Job");
  if ( HNOSrc != ARG_INVALID_HANDLE_VALUE ) {
    char buffer[255];
    if ( LoadTreeFromString(HC, HNODst, buffer, 255) ==
        E_OK ) {
      printf("The node created is: %s\n", buffer);
    }
  }
  FreeSaveTreeString(tmp);
}
```

par **HC** is the handle to the controller

HNO is the handle to the NodeObject from where the string shall be retrieved

ret a string representation of the subtree. The caller of this function is responsible for freeing the memory of that string. NULL, if an error occured or the NodeObject does not exist

see 4.5.6.9 FreeSaveTreeString, 4.5.6.11 LoadTreeFromString, 4.5.6.8 SaveTreeAsXMLString, 4.5.6.5 SaveTree, 4.5.6.3 SaveJob

4.5.6.8. SaveTreeAsXMLString

fct

```
char* SaveTreeAsXMLString(HController HC, HNodeObject HNO);
```

Gets the string representation of a subtree.

Listing 4.15: SaveTreeAsXMLString example

```
HNodeObject HNOSrc = GetNode(HC, "usr.job.Job.Transform1
  ");
if ( HNOSrc != ARG_INVALID_HANDLE_VALUE ) {
  char *tmp = SaveTreeAsXMLString(HC, HNOSrc);
  HNodeObject HNODst = GetNode(HC, "usr.job.Job");
  if ( HNOSrc != ARG_INVALID_HANDLE_VALUE ) {
    char buffer[255];
    if ( LoadTreeFromString(HC, HNODst, buffer, 255) ==
       E_OK ) {
      printf("The node created is: %s\n", buffer);
    }
  }
  FreeSaveTreeString(tmp);
}
```

par **HC** is the handle to the controller

HNO is the handle to the NodeObject from where the string shall be retrieved

ret a string representation of the subtree. The caller of this function is responsible for freeing the memory of that string. `NULL`, if an error occured or the NodeObject does not exist

see 4.5.6.9 FreeSaveTreeString, 4.5.6.7 SaveTreeAsString, 4.5.6.11 LoadTreeFromString, 4.5.6.5 SaveTree, 4.5.6.3 SaveJob

4.5.6.9. FreeSaveTreeString

fct

```
void FreeSaveTreeString(char *string);
```

Frees the string that was retrieved by a call to SaveTreeAsString or SaveTreeAsXMLString.

> **TIP**
>
> This function only has to be used on Windows®. On Linux® and macOS® the application code can free memory, which was alloced to a shared library.

Listing 4.16: FreeSaveTreeString example

```
HNodeObject HNOSrc = GetNode(HC, ”usr.job.Job.Transform1
   ”);
if ( HNOSrc != ARG_INVALID_HANDLE_VALUE ) {
  char *tmp = SaveTreeAsXMLString(HC, HNOSrc);
  HNodeObject HNODst = GetNode(HC, ”usr.job.Job”);
  if ( HNOSrc != ARG_INVALID_HANDLE_VALUE ) {
    char buffer[255];
    if ( LoadTreeFromString(HC, HNODst, buffer, 255) ==
       E_OK ) {
      printf(”The node created is: %s\n”, buffer);
    }
  }
  FreeSaveTreeString(tmp);
}
```

par **`string`** is the string obtained by SaveTreeAsString or SaveTreeAsXMLString

see 4.5.6.7 SaveTreeAsString, 4.5.6.8 SaveTreeAsXMLString

4.5.6.10. LoadTree

fct

```
int LoadTree(HController HC, HNodeObject HNO, const char *
filename);
```

Loads a file, that contains variable information, to the controller. Such files usually are `*.tre` files.

It is possible to load files that were saved by SaveTree or SaveTreeXML.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`filename` full path to the `*.tre` file

ret **`E_OK`** on success

`E_NFILE` if the file was not found or could not be read

`E_FAILURE` on failure

see 4.5.6.7 SaveTreeAsString, 4.5.6.8 SaveTreeAsXMLString

4.5.6.11. LoadTreeFromString

fct

```
int LoadTreeFromString(HController HC, HNodeObject HNO,
const char *str, char *retname, int retnamelen);
```

Loads the string representation of a subtree to the controller. See SaveTreeAsString and SaveTreeAsXMLString for an example.

par **`HC`** is the handle to the controller

`HNO` is the handle to a NodeObject

`str` is the string containing the subtree

`retname` is the buffer for the name of the node that was created by the firmware

retnamelen is the size of the buffer. If the buffer is too small then no name will be returned.

ret **E_OK** on success

E_NFILE if the file was not found or could not be read

E_FAILURE on failure

see 4.5.6.7 SaveTreeAsString, 4.5.6.8 SaveTreeAsXMLString

4.5.6.12. LoadTreeFromStringExt

fct

```
int LoadTreeFromStringExt(HController HC, HNodeObject HNO,
const char *str, int index, char *retname, int retnamelen);
```

Loads the string representation of a subtree to the controller. See SaveTreeAsString and SaveTreeAsXMLString for an example.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

str is the string containing the subtree

index is the index where to put the subtree

retname is the buffer for the name of the node that was created by the firmware

retnamelen is the size of the buffer. If the buffer is too small then no name will be returned

ret **E_OK** on success

E_NFILE if the file was not found or could not be read

E_FAILURE on failure

see 4.5.6.7 SaveTreeAsString, 4.5.6.8 SaveTreeAsXMLString

4.5.6.13. SavePen

fct

```
int SavePen(HController HC, const char *penname, const char
 *filename);
```

Saves a pen to the file system.

par **`HC`** is the handle to the controller

`jobname` is the name of the pen, e.g. `usr.pens.MyPen` or simply `MyPen`

`filename` is the file name of the pen. The path must be accessible from the client application

ret **`E_OK`** on success

`E_FAILURE` on failure

`E_TIMEOUT` if the communication with the firmware times out

see 4.5.6.14 LoadPen, 4.5.6.5 SaveTree

4.5.6.14. LoadPen

fct

```
int LoadPen(HController HC, const char *penname, const char
 *filename);
```

Loads the pen with a given file name to the controller. If the pen name already exists there then the pen on the controller is completly replaced with the new one.

If a pen is uploaded, while a job is running, then `E_BUSY` is returned. If a job is selected then the PLC state `JOB_READY` goes away while loading the pen.

TIP

- *default* and *bitmap* are reserved for internal use. If these names are used as penname the result on the controller is undefined. For piloting a pen named *pilot* has to loaded to the controller
- The upload of a pen is only allowed when no job is running
- The penpath, set with SetPenPath, is not used for this function

par **`HC`** is the handle to the controller

`penname` is the name for the pen on the controller

`filename` is the file name of the pen. The path has to be accessible from the client application

ret **`E_OK`** on success

`E_FAILURE` on failure

`E_BUSY` if a job is running on the controller while the pen is being uploaded

`E_TIMEOUT` if the communication with the firmware times out

`E_NOSPACE` if there is not enough memory for the pen

see 4.5.6.1 LoadJob, 4.5.6.14 LoadPen, 4.5.12.63 RegisterOnPLCChanged

4.5.6.15. LoadFont

fct

```
int LoadFont(HController HC, const char *filename);
```

Loads a TTF- or FDT-font to the controller.

par **`HC`** is the handle to the controller

`filename` is the file name of the TTF- or FDT-font. The path has to be accessible from the client application

ret `E_OK` on success

`E_FAILURE` on failure

`E_TIMEOUT` if the communication with the firmware times out

`E_NOEXIST` if the file does not exist

see 4.5.6.1 LoadJob, 4.5.6.14 LoadPen, 4.5.5.1 SetPenPath, 4.5.12.63 RegisterOnPLCChanged

4.5.6.16. LoadDataXMLFromFile

fct

```
int LoadDataXMLFromFile(HController HC, const char *
filename);
```

Loads XML-data to the controller. Only available if the firmware supports the feature *XMLDataLoad*.

par `HC` is the handle to the controller

`filename` is the file name which holds the XML-data

ret `E_OK` on success

`E_FAILURE` on failure

`E_TIMEOUT` if the communication with the firmware times out

`E_NFILE` if the file does not exist

`E_UNIMP` if the firmware does not support this feature

see 4.5.6.17 LoadDataXML

4.5.6.17. LoadDataXML

fct

```
int LoadDataXML(HController HC, const char *xmldata);
```

Loads XML-data to the controller. Only available if the firmware supports the feature *XMLDataLoad*.

par **HC** is the handle to the controller

xmldata is the the XML-data

ret **E_OK** on success

E_FAILURE on failure

E_TIMEOUT if the communication with the firmware times out

E_UNIMP if the firmware does not support this feature

see 4.5.6.16 LoadDataXMLFromFile

4.5.7. Nodes

Table of contents

4.5.7.1. GetNode . . . 94
4.5.7.2. GetNodeFromCache . . . 95
4.5.7.3. GetNodeInfo . . . 95
4.5.7.4. GetNodeInfoExt . . . 96
4.5.7.5. DestroyNodeInfo . . . 98
4.5.7.6. ReadNode . . . 98
4.5.7.7. WriteNode . . . 99
4.5.7.8. WriteNodeSync . . . 99
4.5.7.9. WriteNodeAndStoreLocal . . . 100
4.5.7.10. GetNodeFlags . . . 100
4.5.7.11. SetNodeFlags . . . 101
4.5.7.12. SetNodeMinMax . . . 102
4.5.7.13. SetNodeUnit . . . 103
4.5.7.14. GetNodeState . . . 104
4.5.7.15. GetParentNode . . . 104
4.5.7.16. GetSubnodesCount . . . 105
4.5.7.17. SelectNode . . . 105
4.5.7.18. SelectNodeByName . . . 106
4.5.7.19. DeselectNode . . . 106
4.5.7.20. DeselectNodeByName . . . 107
4.5.7.21. TeachInSetCurrent . . . 107
4.5.7.22. TeachInSetCurrentByName . . . 108
4.5.7.23. DeleteNode . . . 109
4.5.7.24. DeleteNodeOnController . . . 109
4.5.7.25. DeleteNodeOnControllerByName . . . 110
4.5.7.26. DeleteSingleNodeOnController . . . 111
4.5.7.27. DeleteSingleNodeOnControllerByName . . . 111
4.5.7.28. RenameNode . . . 112
4.5.7.29. MoveSubtree . . . 113

4.5.7.30. CreateNodeOnController 114
4.5.7.31. CreateNodeOnControllerExt 114
4.5.7.32. CreateNodeOnControllerAsync 116
4.5.7.33. CreateJobNodeOnControllerExt 117
4.5.7.34. CreateJobNodeOnController 118
4.5.7.35. CreateJobNodeOnControllerAsync 118
4.5.7.36. GetNodeIndex 119
4.5.7.37. GetNodeType 120
4.5.7.38. GetNodeTypeStringLen 121
4.5.7.39. GetNodeTypeString 122
4.5.7.40. GetNodeID 123
4.5.7.41. GetNodeNameLen 123
4.5.7.42. GetNodeName 124
4.5.7.43. GetNodeLastNameLen 125
4.5.7.44. GetNodeLastName 125
4.5.7.45. ModifyAllowed 126
4.5.7.46. RenameAllowed 127
4.5.7.47. DeleteAllowed 127
4.5.7.48. CreateAllowed 128
4.5.7.49. FlagsModifyAllowed 128
4.5.7.50. ExtendRangeAllowed 129
4.5.7.51. ModifyMinMaxAllowed 129
4.5.7.52. ModifyUnitAllowed 130
4.5.7.53. IsConsumable 130
4.5.7.54. SetConsumable 131
4.5.7.55. IsPenable 131
4.5.7.56. IsControlledByDevice 132
4.5.7.57. SetControlledByDevice 132
4.5.7.58. IsOwnedByDevice 133
4.5.7.59. IsProtected 133
4.5.7.60. IsModified 134
4.5.7.61. IsMirrored 134

4.5.7.62. SetMirrored 135
4.5.7.63. IsQuicksaveable 136
4.5.7.64. SetQuicksaveable 136
4.5.7.65. IsForcePen 137
4.5.7.66. SetForcePen 137
4.5.7.67. IsUserdefined 138
4.5.7.68. IsWriteable 138
4.5.7.69. UserCreatable 139
4.5.7.70. CanMakeLines 139
4.5.7.71. CanPassLines 140
4.5.7.72. AcceptsSubnodes 140
4.5.7.73. GetSelectEntriesCount 141
4.5.7.74. GetSelectEntryLength 141
4.5.7.75. GetSelectEntry 142
4.5.7.76. SetNodeValueBin 143
4.5.7.77. GetNodeValueBinCopyLen 144
4.5.7.78. GetNodeValueBinCopy 145
4.5.7.79. SetNodeValueBool 145
4.5.7.80. GetNodeValueBool 146
4.5.7.81. GetNodeValueStringLen 146
4.5.7.82. GetNodeValueString 147
4.5.7.83. GetNodeValueInt32 148
4.5.7.84. GetNodeValueInt64 149
4.5.7.85. GetNodeValueReal32 150
4.5.7.86. GetNodeValueReal64 150
4.5.7.87. SetNodeValueString 151
4.5.7.88. SetNodeValueInt32 152
4.5.7.89. SetNodeValueInt64 153
4.5.7.90. SetNodeValueReal32 153
4.5.7.91. SetNodeValueReal64 154
4.5.7.92. GetMin 155
4.5.7.93. GetMax 155

4.5.7.94. GetUnitLen 156
4.5.7.95. GetUnit 157
4.5.7.96. GetSubnodesNamesLen 158
4.5.7.97. GetSubnodesNames 158
4.5.7.98. GetSubnodesHNOArray 159
4.5.7.99. IsNodeChildOf 160
4.5.7.100. IsNodeDescendantOf 161
4.5.7.101. GetEditorHint 162

4.5.7.1. GetNode

fct

```
HNodeObject GetNode(HController HC, const char* name);
```

Gets a node for the given variable name, e.g. `usr.job.Job`). If the variable cache was enabled with EnableVariableCache the handle to the variable is returned. If the variable cache was not enabled this function only creates a virtual *NodeObject*. It is not filled with any data until a call to ReadNode is executed. If the ID (GetNodeID) of the node is -1 the node does not exist on the controller.

If the variable cache was enabled the root-node can be get with

```
HNodeObject HRootNode = GetNode(HC, ””);
```

par **`HC`** is the handle to the controller

`name` is the name of the controller variable

ret Handle to a NodeObject

`ARG_INVALID_HANDLE_VALUE` on error

see 4.5.7.2 GetNodeFromCache, 4.5.7.6 ReadNode, 4.5.7.7 WriteNode, 4.5.2.6 EnableVariableCache, 4.5.7.40 GetNodeID, 4.5.7.3 GetNodeInfo

4.5.7.2. GetNodeFromCache

fct

```
HNodeObject GetNodeFromCache(HController HC, const char *
name);
```

If the variable cache was enabled with EnableVariableCache this method returns a handle to this node otherwise it returns `ARG_INVALID_HANDLE_VALUE`. The difference to GetNode is, that GetNode returns a valid handle even if the node does not exist on the controller because if the variable cache was not enabled a call to ReadNode is necessary to find out, if the node exists.

Listing 4.17: GetNodeFromCache example

```
HNodeObject HNO = GetNodeFromCache(HC, "usr.var.x");
if ( HNO != ARG_INVALID_HANDLE_VALUE ) {
  // Node usr.var.x exists.
}
```

par **`HC`** is the handle to the controller

`name` is the name of the controller variable

ret Handle to a NodeObject

`ARG_INVALID_HANDLE_VALUE` if the node dos not exist

see 4.5.7.1 GetNode, 4.5.2.6 EnableVariableCache, 4.5.7.6 ReadNode

4.5.7.3. GetNodeInfo

fct

```
ARG_NODEINFO* GetNodeInfo(HController HC, HNodeObject HNO);
```

Gets information about a NodeObject. See ARG_NODEINFO for a detailed description of the ARG_NODEINFO-structure.

Listing 4.18: GetNodeInfo example

```
HNodeObject HNO = GetNode(HC, "stat.time.TimeStr");
if ( HNO != ARG_INVALID_HANDLE_VALUE ) {
  ARG_NODEINFO *info = GetNodeInfo(HC, HNO);
  if ( info != NULL ) {
    printf("Nodename:  %s\n", info->fullname);
    printf("Value:     %s\n", info->value);
    printf("Unit:      %s\n", info->unit);
    DestroyNodeInfo(info);
  }
}
```

TIP

The decimal separator is a dot by default. To change this to your current locale call DisableFloatingPointToStringDot.

par `HC` is the handle to the controller

`HC` is the handle to the NodeObject

ret the pointer to a filled ARG_NODEINFO-structure or `NULL`)

see 4.1.2.1 ARG_NODEINFO, 4.5.1.6 DisableFloatingPointToStringDot, 4.5.7.4 GetNodeInfoExt, 4.5.7.5 DestroyNodeInfo, 4.5.7.1 GetNode

4.5.7.4. GetNodeInfoExt

fct

```
ARG_NODEINFO* GetNodeInfoExt(HController HC, HNodeObject
HNO, unsigned int flags);
```

Gets information about a NodeObject. See ARG_NODEINFO for a detailed description of the ARG_NODEINFO-structure.

This function performs better than GetNodeInfo in cases when only some informations of the node are needed. Only the "expensive" string-copies can be influenced with this function. The following flags can be passed to this function.

NODEINFO_FULLNAME If this flag is set, the full name will be set in the ARG_NODEINFO-structure.

NODEINFO_NAME If this flag is set, the name will be set in the ARG_NODEINFO-structure.

NODEINFO_PATH If this flag is set, the path will be set in the ARG_NODEINFO-structure.

NODEINFO_VALUE If this flag is set, the value will be set in the ARG_NODEINFO-structure.

NODEINFO_PRIV If this flag is set, the priv will be set in the ARG_NODEINFO-structure.

NODEINFO_UNIT If this flag is set, the unit will be set in the ARG_NODEINFO-structure.

NODEINFO_TYPESTRING If this flag is set, the typestring will be set in the ARG_NODEINFO-structure.

NODEINFO_ALL If this flag is set, all fields will be set.

Listing 4.19: GetNodeInfoExt example

```
HNodeObject HNO = GetNode(HC, "stat.time.TimeStr");
if ( HNO != ARG_INVALID_HANDLE_VALUE ) {
  ARG_NODEINFO *info = GetNodeInfoExt(HC, HNO,
     NODEINFO_FULLNAME | NODEINFO_VALUE);
  if ( info != NULL ) {
    printf("Nodename:  %s\n", info->fullname);
    printf("Value:     %s\n", info->value);
    DestroyNodeInfo(info);
  }
}
```

TIP

The decimal separator is a period (dot) by default. To change this to your current locale call DisableFloatingPointToStringDot.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

ret the pointer to a filled ARG_NODEINFO-structure or `NULL`

see 4.1.2.1 ARG_NODEINFO, 4.5.1.6 DisableFloatingPointToStringDot, 4.5.7.3 GetNodeInfo, 4.5.7.5 DestroyNodeInfo, 4.5.7.1 GetNode

4.5.7.5. DestroyNodeInfo

fct

```
void DestroyNodeInfo(ARG_NODEINFO *info);
```

Destroys a NodeInfo which was allocated with GetNodeInfo.

par **info** is the NodeInfo which was allocated with GetNodeInfo

see 4.5.7.3 GetNodeInfo, 4.5.7.4 GetNodeInfoExt

4.5.7.6. ReadNode

fct

```
int ReadNode(HController HC, HNodeObject HNO);
```

Reads the current value of a controller variable. To get notified when a variable changes on the controller use RegisterOnValueChanged.

TIP

Calls to this function are only needed when the variable cache was not enabled with EnableVariableCache.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

ret **E_OK** on success

E_TIMEOUT if the communication with the firmware times out

E_FAILURE on failure

see 4.5.7.1 GetNode, 4.5.7.7 WriteNode, 4.5.12.10 RegisterOnValueChanged, 4.5.2.6 EnableVariableCache, 4.5.7.3 GetNodeInfo, 4.5.7.4 GetNodeInfoExt

4.5.7.7. WriteNode

fct

```
int WriteNode(HController HC, HNodeObject HNO);
```

Writes the current value of a NodeObject to an associated controller variable. If changing the node forces other nodes to change then WriteNodeSync should be used.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

ret **E_OK** on success

E_TIMEOUT if the communication with the firmware times out

E_FAILURE on failure

see 4.5.7.1 GetNode, 4.5.7.6 ReadNode, 4.5.7.8 WriteNodeSync

4.5.7.8. WriteNodeSync

fct

```
int WriteNodeSync(HController HC, HNodeObject HNO);
```

Writes the current value of a NodeObject to an associated controller variable. This function waits until the firmware has sent all updates. This is useful if setting a variable changes other variables.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

ret **E_OK** on success

E_TIMEOUT if the communication with the firmware times out

E_FAILURE on failure

see 4.5.7.1 GetNode, 4.5.7.6 ReadNode

4.5.7.9. WriteNodeAndStoreLocal

fct
```
int WriteNodeAndStoreLocal(HController HC, HNodeObject HNO);
```

Writes the current value of a NodeObject to an associated controller variable. If changing the node forces other nodes to change then WriteNodeSync should be used.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

ret **E_OK** on success

E_TIMEOUT if the communication with the firmware times out

E_FAILURE on failure

see 4.5.7.1 GetNode, 4.5.7.6 ReadNode, 4.5.7.8 WriteNodeSync

4.5.7.10. GetNodeFlags

fct
```
int GetNodeFlags(HController HC, HNodeObject HNO, unsigned
int *flags);
```

Reads the flags of the given NodeObject. The flags are normally used for debugging issues only.

> **TIP**
>
> With GetNodeInfo it is possible to get all information about a node with a single call.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

flags holds the flags after returning with `E_OK`

ret **E_OK** on success

E_FAILURE on failure

see 4.5.7.1 GetNode, 4.5.12.10 RegisterOnValueChanged, 4.5.2.6 EnableVariableCache, 4.5.7.3 GetNodeInfo, 4.5.7.4 GetNodeInfoExt, 4.5.7.7 WriteNode, 4.5.7.11 SetNodeFlags, 4.5.12.17 RegisterOnFlagsChanged, 4.5.7.48 CreateAllowed, 4.5.7.47 DeleteAllowed, 4.5.7.50 ExtendRangeAllowed, 4.5.7.49 FlagsModifyAllowed, 4.5.7.45 ModifyAllowed, 4.5.7.46 RenameAllowed, 4.5.7.53 IsConsumable, 4.5.7.56 IsControlledByDevice, 4.5.7.65 IsForcePen, 4.5.7.61 IsMirrored, 4.5.7.60 IsModified, 4.5.7.58 IsOwnedByDevice, 4.5.7.55 IsPenable, 4.5.7.59 IsProtected, 4.5.7.63 IsQuicksaveable, 4.5.7.67 IsUserdefined, 4.5.7.68 IsWriteable, 4.5.7.54 SetConsumable, 4.5.7.57 SetControlledByDevice, 4.5.7.66 SetForcePen, 4.5.7.62 SetMirrored, 4.5.7.64 SetQuicksaveable

4.5.7.11. SetNodeFlags

fct

```
int SetNodeFlags(HController HC, HNodeObject HNO, unsigned
int setflags, unsigned int unsetflags);
```

Sets and unsets specific flags for the node. Please note, this function returns immediately. If flags are actually changed then the FlagsChangeCallbackFunction is called.

> **TIP**
>
> Please use this function with special care. Normally there is no need for this function.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

setflags holds the flags that should be set to 1

unsetflags holds the flags that should be set to 0

ret **E_OK** on success

E_FAILURE on failure

see 4.2.1.5 FlagsChangeCallbackFunction
4.2.1.4 FlagsChangeCallbackFunctionExt 4.5.7.10 GetNodeFlags,
4.5.7.48 CreateAllowed, 4.5.7.47 DeleteAllowed,
4.5.7.50 ExtendRangeAllowed, 4.5.7.49 FlagsModifyAllowed,
4.5.7.45 ModifyAllowed, 4.5.7.46 RenameAllowed, 4.5.7.53 IsConsumable,
4.5.7.56 IsControlledByDevice, 4.5.7.65 IsForcePen, 4.5.7.61 IsMirrored,
4.5.7.60 IsModified, 4.5.7.58 IsOwnedByDevice, 4.5.7.55 IsPenable,
4.5.7.59 IsProtected, 4.5.7.63 IsQuicksaveable, 4.5.7.67 IsUserdefined,
4.5.7.68 IsWriteable, 4.5.7.54 SetConsumable,
4.5.7.57 SetControlledByDevice, 4.5.7.66 SetForcePen,
4.5.7.62 SetMirrored, 4.5.7.64 SetQuicksaveable

4.5.7.12. SetNodeMinMax

fct

```
int SetNodeMinMax(HController HC, HNodeObject HNO, float
vmin, float vmax);
```

Sets the minimum and maximum value of a user created variable. If the variable should have no minimum and maximum value then vmax has to be smaller than vmin.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

vmin is the new minimum value of the variable

vmax is the new maximum value of the variable

ret **E_OK** on success

E_UNIMPL if the firmware does not support setting

E_NALLOWED if setting the min/max value of the NodeObject is not allowed

E_FAILURE on failure

see 4.5.7.48 CreateAllowed, 4.5.7.47 DeleteAllowed, 4.5.7.50 ExtendRangeAllowed, 4.5.7.49 FlagsModifyAllowed, 4.5.7.45 ModifyAllowed, 4.5.7.46 RenameAllowed, 4.5.7.53 IsConsumable, 4.5.7.56 IsControlledByDevice, 4.5.7.65 IsForcePen, 4.5.7.61 IsMirrored, 4.5.7.60 IsModified, 4.5.7.58 IsOwnedByDevice, 4.5.7.55 IsPenable, 4.5.7.59 IsProtected, 4.5.7.63 IsQuicksaveable, 4.5.7.67 IsUserdefined, 4.5.7.68 IsWriteable, 4.5.7.54 SetConsumable, 4.5.7.57 SetControlledByDevice, 4.5.7.66 SetForcePen, 4.5.7.62 SetMirrored, 4.5.7.64 SetQuicksaveable

4.5.7.13. SetNodeUnit

fct

```
int SetNodeUnit(HController HC, HNodeObject HNO, const char
 *unit);
```

Sets the unit of a user created variable.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

unit is the new unit of the variable

ret **E_OK** on success

E_UNIMPL if the firmware does not support setting

E_NALLOWED if setting the unit of the NodeObject is not allowed

E_FAILURE on failure

see 4.5.7.48 CreateAllowed, 4.5.7.47 DeleteAllowed, 4.5.7.50 ExtendRangeAllowed, 4.5.7.49 FlagsModifyAllowed, 4.5.7.45 ModifyAllowed, 4.5.7.46 RenameAllowed, 4.5.7.53 IsConsumable, 4.5.7.56 IsControlledByDevice, 4.5.7.65 IsForcePen, 4.5.7.61 IsMirrored, 4.5.7.60 IsModified, 4.5.7.58 IsOwnedByDevice, 4.5.7.55 IsPenable, 4.5.7.59 IsProtected, 4.5.7.63 IsQuicksaveable, 4.5.7.67 IsUserdefined, 4.5.7.68 IsWriteable, 4.5.7.54 SetConsumable, 4.5.7.57 SetControlledByDevice, 4.5.7.66 SetForcePen, 4.5.7.62 SetMirrored, 4.5.7.64 SetQuicksaveable

4.5.7.14. GetNodeState

fct

```
int GetNodeState(HController HC, HNodeObject HNO);
```

Gets the current state of the state machine in the NodeObject. For supported states see NodeObject states.

par **HC** is the handle to the controller

HNO is handle to the NodeObject

ret the state of the NodeObject

see 4.3.1 NodeObject states, 4.2.1.19 NodeStateChangeCallbackFunction

4.5.7.15. GetParentNode

fct

```
HNodeObject GetParentNode(HController HC, HNodeObject HNO);
```

Gets the parent of a node if this parent is also loaded by the ControllerLib.

> **TIP**
>
> Only works correctly if the VariableCache was enabled with EnableVariableCache.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

ret the handle to the parent node

ARG_INVALID_HANDLE_VALUE if not loaded by the library

see 4.5.2.6 EnableVariableCache, 4.5.8.6 GetSubnodes, 4.5.7.1 GetNode

4.5.7.16. GetSubnodesCount

fct
```
int GetSubnodesCount(HController HC, HNodeObject HNO);
```

Gets the number of children of a node.

TIP

Only works correctly when the VariableCache was enabled with EnableVariableCache.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

ret the number of subnodes of a NodeObject

E_FAILURE on failure

see 4.5.2.6 EnableVariableCache, 4.5.8.6 GetSubnodes, 4.5.7.1 GetNode

4.5.7.17. SelectNode

fct
```
int SelectNode(HController HC, HNodeObject HNO);
```

Selects a node for job execution on the controller.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

ret **E_OK** on success

E_TIMEOUT if the communication with the firmware times out

E_FAILURE on failure

see 4.5.7.18 SelectNodeByName, 4.5.7.19 DeselectNode, 4.5.6.1 LoadJob, 4.5.4.2 JobStart, 4.5.12.63 RegisterOnPLCChanged

4.5.7.18. SelectNodeByName

fct

```
int SelectNodeByName(HController HC, const char *varname);
```

Selects a node for job execution on the controller by path name.

par **HC** is the handle to the controller

varname is the complete path to the node to be selected

ret **E_OK** on success

E_TIMEOUT if the communication with the firmware times out

E_FAILURE on failure

see 4.5.7.17 SelectNode, 4.5.7.20 DeselectNodeByName, 4.5.6.1 LoadJob, 4.5.4.2 JobStart, 4.5.12.63 RegisterOnPLCChanged

4.5.7.19. DeselectNode

fct

```
int DeselectNode(HController HC, HNodeObject HNO);
```

Deselects the given NodeObject on the controller.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

ret **E_OK** on success

E_TIMEOUT if the communication with the firmware times out

E_FAILURE on failure

see 4.5.7.20 DeselectNodeByName, 4.5.7.17 SelectNode, 4.5.6.1 LoadJob, 4.5.4.2 JobStart, 4.5.12.63 RegisterOnPLCChanged

4.5.7.20. DeselectNodeByName

fct

```
int DeselectNodeByName(HController HC, const char *varname);
```

Deselects a node on the controller by its full qualified name.

par **HC** is the handle to the controller

varname is the fully qualified name of the node to be deselected

ret **E_OK** on success

E_TIMEOUT if the communication with the firmware times out

E_FAILURE on failure

see 4.5.7.19 DeselectNode, 4.5.7.17 SelectNode, 4.5.6.1 LoadJob, 4.5.4.2 JobStart, 4.5.12.63 RegisterOnPLCChanged

4.5.7.21. TeachInSetCurrent

fct

```
int TeachInSetCurrent(HController HC, HNodeObject HNO);
```

Sets the node to be marked in teach-in mode to the given NodeObject. To find out which node is the current TeachInNode, the `VAR:STRING` variable `dev.sas.TeachIn.CurrentTeachInNode` holds the name.

> **TIP**
>
> For teach-in to work properly, a job on the controller must be selected. The PLC states *JOB_READY* and *DEVICES_READY* have to be present and a pen named *pilot* has to be on the controller.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

ret **E_OK** on success

E_TIMEOUT if the communication with the firmware times out

E_FAILURE on failure

see 4.5.7.22 TeachInSetCurrentByName, 4.5.4.3 JobPilot

4.5.7.22. TeachInSetCurrentByName

fct

```
int TeachInSetCurrentByName(HController HC, const char *
varname);
```

Sets the node to be marked in teach-in mode to the given NodeObject by path name. To find out which node is the current TeachInNode the `VAR:STRING` variable `dev.sas.TeachIn.CurrentTeachInNode` can be read.

TIP

For teach-in to work properly, a job on the controller must be selected. The PLC states *JOB_READY* and *DEVICES_READY* have to be present and a pen name *pilot* has to be on the controller.

par **HC** is the handle to the controller

varname is the complete name of the variable to be marked in teach-in mode

ret **E_OK** on success

E_TIMEOUT if the communication with the firmware times out

E_FAILURE on failure

see 4.5.7.21 TeachInSetCurrent, 4.5.4.3 JobPilot

4.5.7.23. DeleteNode

fct
```
int DeleteNode(HController HC, HNodeObject HNO);
```

Deletes the *connection* to a variable on the controller. This call *does not delete* the variable from the controller and is only necessary if the variable cache was not enabled with EnableVariableCache.

> **TIP**
>
> To delete a variable on the controller use DeleteNodeOnController or DeleteNodeOnControllerByName.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

ret `E_OK` on success

`E_NALLOWED` if the variable cache was enabled

`E_FAILURE` on failure

see 4.5.7.1 GetNode, 4.5.2.6 EnableVariableCache,
4.5.7.24 DeleteNodeOnController,
4.5.7.25 DeleteNodeOnControllerByName,
4.5.7.26 DeleteSingleNodeOnController,
4.5.7.27 DeleteSingleNodeOnControllerByName

4.5.7.24. DeleteNodeOnController

fct
```
int DeleteNodeOnController(HController HC, HNodeObject HNO);
```

Deletes a node including its subnodes from the controller. The node has to be identified by its handle. If the call was successfull then `E_OK` is returned. If the node did not exist on the controller, the call returns also with `E_OK`, because the call itself was successfull, but a SysMessage will be thrown by the controller.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

ret **E_OK** on success

E_FAILURE on failure

see 4.5.7.25 DeleteNodeOnControllerByName,
4.5.7.27 DeleteSingleNodeOnControllerByName,
4.5.7.26 DeleteSingleNodeOnController,
4.5.7.30 CreateNodeOnController, 4.5.7.47 DeleteAllowed

4.5.7.25. DeleteNodeOnControllerByName

fct

```
int DeleteNodeOnControllerByName(HController HC, const char
 *nodename);
```

Deletes a node from the controller. The node has to be identified by its name. If the call was successfull then `E_OK` is returned. If the node did not exist on the controller then the call returns also with `E_OK`, because the call itself was successfull, but a SysMessage will be thrown by the controller.

> **TIP**
>
> The node including all subnodes is deleted on the controller.

par **HC** is the handle to the controller

nodename is the full name of the variable; e.g. `usr.var.myvar`

ret **E_OK** on success

E_FAILURE on failure

see 4.5.7.24 DeleteNodeOnController,
4.5.7.27 DeleteSingleNodeOnControllerByName,
4.5.7.26 DeleteSingleNodeOnController,
4.5.7.30 CreateNodeOnController, 4.5.7.47 DeleteAllowed

4.5.7.26. DeleteSingleNodeOnController

fct

```
int DeleteSingleNodeOnController(HController HC,
HNodeObject HNO);
```

> ⚠ **NOTICE**
>
> This function is deprecated.
>
> - Use DeleteNodeOnController or DeleteNodeOnControllerByName instead.

The description of this function has intentionally been omitted.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

ret **`E_NOEXIST`**

see 4.5.7.27 DeleteSingleNodeOnControllerByName, 4.5.7.25 DeleteNodeOnControllerByName, 4.5.7.24 DeleteNodeOnController, 4.5.7.30 CreateNodeOnController, 4.5.7.47 DeleteAllowed

4.5.7.27. DeleteSingleNodeOnControllerByName

fct

```
int DeleteSingleNodeOnControllerByName(HController HC,
const char *nodename);
```

> ⚠ **NOTICE**
>
> This function is deprecated.
>
> - Use DeleteNodeOnController or DeleteNodeOnControllerByName instead.

The description of this function has intentionally been omitted.

par **HC** is the handle to the controller

nodename is the full name of the variable; e.g. `usr.var.myvar`

ret **E_NOEXIST**

see 4.5.7.26 DeleteSingleNodeOnController,
4.5.7.25 DeleteNodeOnControllerByName,
4.5.7.24 DeleteNodeOnController, 4.5.7.30 CreateNodeOnController,
4.5.7.47 DeleteAllowed

4.5.7.28. RenameNode

fct

```
int RenameNode(HController HC, HNodeObject HNO, const char
*newname);
```

Renames a node on the controller. This function returns immediately. When the name changed on the controller then this can be followed with a NameChangeCallbackFunction.

TIP

- This only changes the last name of the variable.
- To move a node to another subtree use MoveSubtree.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

newname the new last name of the node

ret **E_OK** on success

E_NALLOWED if renaming is not allowed

E_TIMEOUT if the communication with the firmware times out

E_FAILURE on failure

see 4.5.12.53 RegisterOnNameChanged, 4.5.7.29 MoveSubtree

4.5.7.29. MoveSubtree

fct

```
int MoveSubtree(HController HC, HNodeObject HNO,
HNodeObject target, int index);
```

Moves a subtree to the target NodeObject with the given index. This function returns immediately. When the node is moved on the controller this can be followed with a NodeMovedCallbackFunction.

TIP

- Even when this function returns with `E_OK` it is possible that moving the node fails.
- To rename a node only use RenameNode.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject to be moved

`target` is the handle to the target NodeObject

`index` is the new index of the node

ret **`E_OK`** on success

`E_TIMEOUT` if the communication with the firmware times out

`E_NALLOWED` if moving is not allowed

`E_FAILURE` on failure

see 4.5.12.20 RegisterOnNodeMoved, 4.5.7.28 RenameNode

4.5.7.30. CreateNodeOnController

fct

```
int CreateNodeOnController(HController HC, const char *
nodename, VAR_TYPE vartype, int index);
```

Creates a node on the controller. When this function returns the node on the controller is already created.

> **TIP**
>
> It may not be allowed to create a node on most of the subtrees. Normally it is only advisable to do this in `usr.var`.

par **`HC`** is the handle to the controller

`nodename` is the full name of a variable, e.g. `usr.var.myvar`

`nodetype` is the type of the new variable; see GetNodeType for a list of node types

`index` is the index in the tree. Only in special cases, like job trees, this is other than `0`

ret **`E_OK`** on success

`E_TIMEOUT` if the communication with the firmware times out

`E_FAILURE` on failure

see 4.5.7.34 CreateJobNodeOnController, 4.5.7.48 CreateAllowed, 4.5.7.25 DeleteNodeOnControllerByName, 4.5.7.37 GetNodeType, 4.5.7.32 CreateNodeOnControllerAsync

4.5.7.31. CreateNodeOnControllerExt

fct

```
int CreateNodeOnControllerExt(HController HC, const char *
nodename, VAR_TYPE vartype, int index, char *retname, int
retnamelen);
```

Creates a node on the controller. When this function returns the node on the controller is already created and `retname` is filled with the name that the controller returned.

> **TIP**
>
> It may not be allowed to create a node on most of the subtrees. Normally it is only advisable to do this in `usr.var`.

par **`HC`** is the handle to the controller

`nodename` is the full name of the variable, e.g. `usr.var.myvar`

`vartype` is the type of the new variable; see GetNodeType for a list of node types

`index` is the index in the tree. Only in special cases, like job trees, this is other than `0`

`retname` is the place where the ControllerLib can store the name that the firmware returned

`retnamelen` is the length of the buffer for retname. This must be at least 32 characters. If it is smaller then `E_NOSPACE` is returned.

ret **`E_OK`** on success

`E_NOSPACE` if the buffer for retname is too small

`E_TIMEOUT` if the communication with the firmware times out

`E_FAILURE` on failure

see 4.5.7.34 CreateJobNodeOnController, 4.5.7.48 CreateAllowed, 4.5.7.25 DeleteNodeOnControllerByName, 4.5.7.37 GetNodeType, 4.5.7.32 CreateNodeOnControllerAsync

4.5.7.32. CreateNodeOnControllerAsync

fct

```
int CreateNodeOnControllerAsync(HController HC, const char
*nodename, VAR_TYPE vartype, int index);
```

Creates a node on the controller. This call returns immediately and does not wait for notification from the controller. When the node on the controller is actually created this can be catched in a NodeCreatedCallbackFunction or in a NodeCreatedCallbackFunctionExt. If a lot of nodes have to be created this function should be used because it is faster than CreateNodeOnController.

TIP

It may not be allowed to create a node on most of the subtrees. Normally it is only advisable to do this in `usr.var`.

par **`HC`** is the handle to the controller

`nodename` is the full name of a variable, e.g. `usr.var.myvar`

`vartype` is the type of the new variable; see GetNodeType for a list of node types

`index` is the index in the tree. Only in special cases, like job trees, this is other than `0`

ret **`E_OK`** on success

`E_TIMEOUT` if the communication with the firmware times out

`E_FAILURE` on failure

see 4.5.7.34 CreateJobNodeOnController, 4.5.7.48 CreateAllowed,
4.5.7.25 DeleteNodeOnControllerByName, 4.5.7.37 GetNodeType,
4.5.7.30 CreateNodeOnController, 4.2.1.9 NodeCreatedCallbackFunction,
4.2.1.8 NodeCreatedCallbackFunctionExt

4.5.7.33. CreateJobNodeOnControllerExt

fct

```
int CreateJobNodeOnControllerExt(HController HC, const char
 *nodename, const char *jobnodetype, int index, char *
retname, int retnamelen);
```

Creates a job node on the controller. When this function returns the node on the controller is already created and `retname` is filled with the name that the controller returned.

> **TIP**
>
> It may not be allowed to create a node on most of the subtrees. Normally it is only advisable to do this in `usr.var`.

par **`HC`** is the handle to the controller

`nodename` is the full name of the variable, e.g. `usr.job.Job.Shape`

`jobnodetype` e.g. `SHAPE`, `JOB:SHAPE`; the leading `JOB:` is optional

`index` is the index in the tree. Only in special cases, like job trees, this is other than `0`

`retname` is the place where the ControllerLib can store the name the firmware returned

`retnamelen` Length of the buffer for `retname`. This must be at least 32 characters. If it is smaller then `E_NOSPACE` is returned.

ret **`E_OK`** on success

`E_NOSPACE` if the buffer for retname is too small

`E_TIMEOUT` if the communication with the firmware times out

`E_FAILURE` on failure

see 4.5.7.30 CreateNodeOnController, 4.5.7.48 CreateAllowed, 4.5.7.25 DeleteNodeOnControllerByName, 4.5.7.37 GetNodeType

4.5.7.34. CreateJobNodeOnController

fct

```
int CreateJobNodeOnController(HController HC, const char *
nodename, const char *jobnodetype, int index);
```

Creates a job node on the controller.

TIP

It may not be allowed to create a node on most of the subtrees. Normally it is only advisable to do this in `usr.var`.

par **`HC`** is the handle to the controller

`nodename` is the full name of a variable, e.g. `usr.job.Job.Shape`

`jobnodetype` is the job node type, e.g. `SHAPE`, `JOB:SHAPE`; the leading `JOB:` is optional

`index` is the index in the tree. Only in special cases, like job trees, this is other than `0`

ret **`E_OK`** on success

`E_TIMEOUT` if the communication with the firmware times out

`E_FAILURE` on failure

see 4.5.7.30 CreateNodeOnController, 4.5.7.48 CreateAllowed, 4.5.7.25 DeleteNodeOnControllerByName, 4.5.7.37 GetNodeType

4.5.7.35. CreateJobNodeOnControllerAsync

fct

```
int CreateJobNodeOnControllerAsync(HController HC, const
char *nodename, const char *jobnodetype, int index);
```

Creates a job node on the controller. This call returns immediately and does not wait for notification from the controller. When the node on the controller is actually created this can be catched in a NodeCreatedCallbackFunction or in a NodeCreatedCallbackFunctionExt. If a lot of nodes have to be created then this function should be used because it is faster than CreateNodeOnController.

> **TIP**
>
> It may not be allowed to create a node on most of the subtrees. Normally it is only advisable to do this in `usr.var`.

par **`HC`** is the handle to the controller

`nodename` is the full name of the variable, e.g. `usr.job.Job.Shape`

`jobnodetype` is the job node type, e.g. `SHAPE`, `JOB:SHAPE`; the leading `JOB:` is optional

`index` is the index in the tree. Only in special cases, like job trees, this is other than `0`

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.7.30 CreateNodeOnController, 4.5.7.48 CreateAllowed,
4.5.7.25 DeleteNodeOnControllerByName, 4.5.7.37 GetNodeType,
4.2.1.9 NodeCreatedCallbackFunction,
4.2.1.8 NodeCreatedCallbackFunctionExt

4.5.7.36. GetNodeIndex

fct

```
int GetNodeIndex(HController HC, HNodeObject HNO, int *
index);
```

Gets the index of NodeObject. This index shows how to order the NodeObjects on a given level. The sorting is ascending. The order of nodes only matters with the execution of job nodes.

TIP

- When the index changes this can be caught with a NodeMovedCallbackFunction.
- With GetNodeInfo it is possible to get all information about a node with a single call.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`index` holds the index after returning with `E_OK`

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.7.3 GetNodeInfo, 4.5.12.20 RegisterOnNodeMoved, 4.2.1.17 NodeMovedCallbackFunction

4.5.7.37. GetNodeType

fct

```
int GetNodeType(HController HC, HNodeObject HNO);
```

Gets the type of the `HNO`.

TIP

- You have to call ReadNode in the first place to get a valid type information.
- The type of a variable can not change.
- This call works only with variables but not with job nodes. Use GetNodeTypeString to get the type for these job nodes.
- With GetNodeInfo it is possible to get all information about a node with a single call.

par **`HC`** is the handle to the controller

HNO is the handle to the NodeObject

ret **VT_INV** unknown type

VT_STR string

VT_TEXT also string

VT_SEL selection

VT_SET set (can have children nodes)

VT_BOOLEAN boolean

VT_BIN binary

VT_I32 32-bit-integer

VT_I64 64-bit-integer

VT_R32 32-bit-real (float)

VT_R64 64-bit-real (double)

E_FAILURE on failure

see 4.5.7.3 GetNodeInfo, 4.5.7.39 GetNodeTypeString

4.5.7.38. GetNodeTypeStringLen

fct

```
int GetNodeTypeStringLen(HController HC, HNodeObject HNO,
int *length);
```

Gets the length for the type string without the trailing 0-byte.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

length holds the strlen of the node type after returning with `E_OK`

ret **E_OK** on success

E_FAILURE on failure

see 4.5.7.39 GetNodeTypeString, 4.5.7.37 GetNodeType, 4.5.7.3 GetNodeInfo

4.5.7.39. GetNodeTypeString

fct

```
int GetNodeTypeString(HController HC, HNodeObject HNO, char
 *buffer, int len);
```

Gets the type of the `HNO` as string. This works on all available nodes.

TIP

- You have to call ReadNode in the first place to get a valid type information.
- The type of a variable can not change.
- With GetNodeInfo it is possible to get all information about a node with a single call.

Listing 4.20: GetNodeTypeString example

```
int buflen;
if ( GetNodeTypeStringLen(HC, HNO, &buflen) == E_OK ) {
  char *buffer = (char*)malloc(++buflen);
  if (GetNodeTypeString(HC, HNO, buffer, buflen) == E_OK
     ) {
    printf("NodeType: %s\n", buffer);
  }
  free(buffer);
}
```

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`buffer` buffer for the NodeType-string

`len` is the length of the buffer (in bytes)

ret **`E_OK`** on success

E_NOSPACE if the buffer is too small

E_FAILURE on failure

see 4.5.7.38 GetNodeTypeStringLen, 4.5.7.37 GetNodeType, 4.5.7.3 GetNodeInfo

4.5.7.40. GetNodeID

fct

```
int64 GetNodeID(HController HC, HNodeObject HNO);
```

Gets the 64-bit-ID of a `HNO` on the controller. Normally it is not necessary to know this ID in a client application. A variable has this ID over its complete lifetime.

> **TIP**
>
> With GetNodeInfo it is possible to get all information about a node with a single call.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

ret ID of the node

-1 if the ID is unknown

see 4.5.7.3 GetNodeInfo

4.5.7.41. GetNodeNameLen

fct

```
int GetNodeNameLen(HController HC, HNodeObject HNO);
```

Gets the length of the full node name of a NodeObject.

> **TIP**
>
> The length is returned without the trailing 0-byte.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

ret the strlen of the name

`E_FAILURE` on failure

see 4.5.7.42 GetNodeName, 4.5.7.3 GetNodeInfo

4.5.7.42. GetNodeName

fct

```
int GetNodeName(HController HC, HNodeObject HNO, char *
buffer, int len);
```

Copies the full node name to the buffer, e.g. `stat.time.TimeStr`. How to determine the size of the buffer is explained at GetNodeNameLen.

Listing 4.21: GetNodeName example

```
int buflen = GetNodeNameLen(HC, HNO);
if (buflen > 0) {
  char *buffer = (char*)malloc(++buflen);
  if (GetNodeName(HC, HNO, buffer, buflen) == E_OK) {
    printf("NodeName: %s\\n", buffer);
  }
  free(buffer);
}
```

> **TIP**
>
> With GetNodeInfo it is possible to get all information about a node with a single call.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

`buffer` is the buffer for the name of the NodeObject

`len` it the length of the buffer in bytes

ret `E_OK` on success

`E_NOSPACE` if the buffer is too small

`E_FAILURE` on failure

see 4.5.7.41 GetNodeNameLen, 4.5.7.44 GetNodeLastName,
4.5.7.3 GetNodeInfo

4.5.7.43. GetNodeLastNameLen

fct

```
int GetNodeLastNameLen(HController HC, HNodeObject HNO);
```

Gets the length of the last node name of a NodeObject.

> **TIP**
>
> The length is returned without the trailing 0-byte.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

ret the strlen of the name

`E_FAILURE` on failure

see 4.5.7.44 GetNodeLastName, 4.5.7.42 GetNodeName, 4.5.7.3 GetNodeInfo

4.5.7.44. GetNodeLastName

fct

```
int GetNodeLastName(HController HC, HNodeObject HNO, char *
buffer, int len);
```

Copies the last node name to the buffer; e.g. `TimeStr`. How to determine the size of the buffer is explained at GetNodeLastNameLen.

Listing 4.22: GetNodeLastName example

```
int buflen = GetNodeLastNameLen(HC, HNO);
if (buflen > 0) {
  char *buffer = (char*)malloc(++buflen);
  if (GetNodeLastName(HC, HNO, buffer, buflen) == E_OK)
     {
    printf("Last-NodeName: %s\\n", buffer);
  }
  free(buffer);
}
```

TIP

With GetNodeInfo it is possible to get all information about a node with a single call.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

buffer is the buffer for the name of the NodeObject

len is the length of the buffer in bytes

ret **E_OK** on success

E_NOSPACE if the buffer is too small

E_FAILURE on failure

see 4.5.7.41 GetNodeNameLen, 4.5.7.44 GetNodeLastName, 4.5.7.3 GetNodeInfo

4.5.7.45. ModifyAllowed

fct

```
int ModifyAllowed(HController HC, HNodeObject HNO, int *
allowed);
```

Checks, whether it is allowed to modify the value of the NodeObject. This function should be called before any modification to the value of the NodeObject.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

`allowed` holds `1` if allowed and `0` otherwise after returning with `E_OK`

ret `E_OK` on success

`E_FAILURE` on failure

4.5.7.46. RenameAllowed

fct

```
int RenameAllowed(HController HC, HNodeObject HNO, int *
allowed);
```

Checks, whether it is allowed to rename the NodeObject.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

`allowed` holds `1` if allowed and `0` otherwise after returning with `E_OK`

ret `E_OK` on success

`E_FAILURE` on failure

4.5.7.47. DeleteAllowed

fct

```
int DeleteAllowed(HController HC, HNodeObject HNO, int *
allowed);
```

Checks, whether it is allowed to delete the NodeObject.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

allowed holds 1 if allowed and 0 otherwise after returning with E_OK

ret **E_OK** on success

E_FAILURE on failure

see 4.5.7.25 DeleteNodeOnControllerByName

4.5.7.48. CreateAllowed

fct

```
int CreateAllowed(HController HC, HNodeObject HNO, int *
allowed);
```

Checks, whether it is allowed to create subnodes in the NodeObject.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

allowed holds 1 if allowed and 0 otherwise after returning with E_OK

ret **E_OK** on success

E_FAILURE on failure

see 4.5.7.30 CreateNodeOnController

4.5.7.49. FlagsModifyAllowed

fct

```
int FlagsModifyAllowed(HController HC, HNodeObject HNO, int
 *allowed);
```

Checks, whether it is allowed to modify the flags of the NodeObject.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

allowed holds 1 if allowed and 0 otherwise after returning with E_OK

ret **E_OK** on success

E_FAILURE on failure

4.5.7.50. ExtendRangeAllowed

fct

```
int ExtendRangeAllowed(HController HC, HNodeObject HNO, int
 *allowed);
```

Checks, whether it is allowed to extend `VAR:SELECT`-variables by an user entry.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

allowed holds `1` if allowed and `0` otherwise after returning with `E_OK`

ret **E_OK** on success

E_FAILURE on failure

4.5.7.51. ModifyMinMaxAllowed

fct

```
int ModifyMinMaxAllowed(HController HC, HNodeObject HNO,
int *allowed);
```

Checks, whether it is allowed to modify the minimum value and the maximum value of the NodeObject.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

allowed holds `1` if allowed and `0` otherwise after returning with `E_OK`

ret **E_OK** on success

E_FAILURE on failure

4.5.7.52. ModifyUnitAllowed

fct

```
int ModifyUnitAllowed(HController HC, HNodeObject HNO, int
*allowed);
```

Checks, whether it is allowed to modify the unit of the NodeObject.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`allowed` holds `1` if allowed and `0` otherwise after returning with `E_OK`

ret **`E_OK`** on success

`E_FAILURE` on failure

4.5.7.53. IsConsumable

fct

```
int IsConsumable(HController HC, HNodeObject HNO, int *
consumable);
```

Checks, whether the NodeObject is consumable.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`consumable` holds `1` if consumable and `0` otherwise after returning with `E_OK`

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.7.54 SetConsumable, 4.5.7.67 IsUserdefined

4.5.7.54. SetConsumable

fct

```
int SetConsumable(HController HC, HNodeObject HNO, int
consumable);
```

Sets the NodeObject as `consumable`.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

`consumable` set this to `1` if the NodeObject shall be consumable and `0` otherwise

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.7.53 IsConsumable, 4.5.7.49 FlagsModifyAllowed

4.5.7.55. IsPenable

fct

```
int IsPenable(HController HC, HNodeObject HNO, int *penable
);
```

Checks, whether the consumable NodeObject can be used in a pen.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

`penable` holds `1` if penable and `0` otherwise after returning with `E_OK`

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.7.65 IsForcePen

4.5.7.56. IsControlledByDevice

fct

```
int IsControlledByDevice(HController HC, HNodeObject HNO,
int *controlled);
```

Checks, whether the NodeObject is controlled by a device.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

controlled holds `1` if controlled by a device and `0` otherwise after returning with `E_OK`

ret **E_OK** on success

E_FAILURE on failure

see 4.5.7.57 SetControlledByDevice

4.5.7.57. SetControlledByDevice

fct

```
int SetControlledByDevice(HController HC, HNodeObject HNO,
int controlled);
```

Sets the NodeObject as controlled by a device.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

controlled set this to `1` if the NodeObject shall be controlled by a device and `0` otherwise

ret **E_OK** on success

E_FAILURE on failure

see 4.5.7.56 IsControlledByDevice, 4.5.7.49 FlagsModifyAllowed

4.5.7.58. IsOwnedByDevice

fct

```
int IsOwnedByDevice(HController HC, HNodeObject HNO, int *
owned);
```

Checks, whether the NodeObject is owned by a device.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`owned` holds `1` if owned by a device and `0` otherwise after returning with `E_OK`

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.7.56 IsControlledByDevice

4.5.7.59. IsProtected

fct

```
int IsProtected(HController HC, HNodeObject HNO, int *prot);
```

Checks, whether the NodeObject is protected. This means that the controller does not allow the variable to be changed by the user.

> **TIP**
>
> Please use ModifyAllowed to find out whether a variable can be modified or not, because it is not safe to rely on this flag only.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`prot` holds `1` if the NodeObject is protected and `0` otherwise after returning with `E_OK`

ret **`E_OK`** on success

E_FAILURE on failure

see 4.5.7.45 ModifyAllowed, 4.5.7.68 IsWriteable

4.5.7.60. IsModified

fct

```
int IsModified(HController HC, HNodeObject HNO, int *
modified);
```

Checks, whether the NodeObject has been modified. If the NodeObject has the modified-flag then its value differs from the value in the NVRAM.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

modified holds `1` if modified and `0` otherwise after returning with `E_OK`

ret **E_OK** on success

E_FAILURE on failure

see 4.5.4.7 WriteToNVRAM, 4.5.7.61 IsMirrored

4.5.7.61. IsMirrored

fct

```
int IsMirrored(HController HC, HNodeObject HNO, int *
mirrored);
```

Checks, whether the NodeObject is mirrored in the NVRAM. This means that the value can be saved and is automatically restored on controller startup.

> **TIP**
>
> To actually save mirrored NodeObjects to the NVRAM use Write-ToNVRAM. If NodeObjects shall always write their current value then see SetQuicksaveable.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

`mirrored` holds `1` if mirrored in NVRAM and `0` otherwise after returning with `E_OK`

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.7.62 SetMirrored, 4.5.7.63 IsQuicksaveable, 4.5.4.7 WriteToNVRAM

4.5.7.62. SetMirrored

fct

```
int SetMirrored(HController HC, HNodeObject HNO, int
mirrored);
```

Sets the NodeObject to be mirrored in the NVRAM. If a NodeObject is mirrored in the NVRAM then the current value can be saved with a call to WriteToNVRAM and the value will be automatically restored on controller startup.

> **TIP**
>
> To actually save mirrored NodeObjects to the NVRAM use Write-ToNVRAM. If NodeObjects shall always write their current value without the need of WriteToNVRAM then see SetQuicksaveable.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

`mirrored` set this to `1` if the NodeObject shall be mirrored and `0` otherwise

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.7.61 IsMirrored, 4.5.7.63 IsQuicksaveable, 4.5.4.7 WriteToNVRAM, 4.5.7.49 FlagsModifyAllowed

4.5.7.63. IsQuicksaveable

fct

```
int IsQuicksaveable(HController HC, HNodeObject HNO, int *
quicksaveable);
```

Checks, whether the NodeObject is quicksaveable. This means, when the value of the NodeObject changes then the new value will automatically be written to the NVRAM without the need of calling WriteToNVRAM.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`quicksaveable` holds `1` if the NodeObject can be quicksaved and `0` otherwise after returning with `E_OK`

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.7.64 SetQuicksaveable, 4.5.7.61 IsMirrored, 4.5.4.7 WriteToNVRAM

4.5.7.64. SetQuicksaveable

fct

```
int SetQuicksaveable(HController HC, HNodeObject HNO, int
quicksaveable);
```

Sets the NodeObject as quicksaveable. This means, when the value of the NodeObject changes then the new value will automatically be written to the NVRAM without the need of calling WriteToNVRAM.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`quicksaveable` set this to `1` if the NodeObject shall be quicksaveable and `0` otherwise

ret **`E_OK`** on success

E_FAILURE on failure

see 4.5.7.63 IsQuicksaveable, 4.5.7.62 SetMirrored, 4.5.4.7 WriteToNVRAM, 4.5.7.49 FlagsModifyAllowed

4.5.7.65. IsForcePen

fct

```
int IsForcePen(HController HC, HNodeObject HNO, int *force);
```

Checks, whether a NodeObject in a pen forces the pen-mechanism to change immediately when the pen is activated. Otherwise the change would not apply until the value is needed for the first time during job execution.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

force holds `1` if the pen is forced and `0` otherwise after returning with E_OK

ret **E_OK** on success

E_FAILURE on failure

see 4.5.7.66 SetForcePen

4.5.7.66. SetForcePen

fct

```
int SetForcePen(HController HC, HNodeObject HNO, int force);
```

Sets the NodeObject in a pen to force the pen-mechanism to change immediately once the pen is activated. Otherwise the change would not apply until the value is needed for the first time during job execution.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

force set this to 1 if the pen shall be forced and 0 otherwise

ret **E_OK** on success

E_FAILURE on failure

see 4.5.7.65 IsForcePen, 4.5.7.49 FlagsModifyAllowed

4.5.7.67. IsUserdefined

fct

```
int IsUserdefined(HController HC, HNodeObject HNO, int *
userdefined);
```

Checks, whether the NodeObject was created by the user.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

userdefined holds 1 if created by the user and 0 otherwise after returning with E_OK

ret **E_OK** on success

E_FAILURE on failure

4.5.7.68. IsWriteable

fct

```
int IsWriteable(HController HC, HNodeObject HNO, int *
writeable);
```

Checks, whether the NodeObject can be written by the user. In difference to IsProtected this flag can be set by the user.

> **TIP**
>
> Please use ModifyAllowed to evaluate whether a variable can be modified or not, because it is not safe to rely on this flag only.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

writeable holds 1 if writable and 0 otherwise after returning with E_OK

ret **E_OK** on success

E_FAILURE on failure

see 4.5.7.45 ModifyAllowed

4.5.7.69. UserCreatable

fct

```
int UserCreatable(HController HC, HNodeObject HNO, int *
allowed);
```

Checks, whether the NodeObject is a job-node which can be created by the user, like JOB:POLYGON.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

allowed holds 1 if it can be created by the user and
texttt0 otherwise after returning with E_OK

ret **E_OK** on success

E_FAILURE on failure

4.5.7.70. CanMakeLines

fct

```
int CanMakeLines(HController HC, HNodeObject HNO, int *
allowed);
```

Checks, whether the NodeObject is a job-node which outputs lines, like JOB:POLYGON.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

allowed holds 1 if it outputs lines and 0 otherwise after returning with E_OK

ret **E_OK** on success

E_FAILURE on failure

4.5.7.71. CanPassLines

fct

```
int CanPassLines(HController HC, HNodeObject HNO, int *
allowed);
```

Checks, whether the NodeObject is a job-node which can pass lines, like JOB:REPEAT.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

allowed holds 1 if it can pass lines and 0 otherwise after returning with E_OK

ret **E_OK** on success

E_FAILURE on failure

4.5.7.72. AcceptsSubnodes

fct

```
int AcceptsSubnodes(HController HC, HNodeObject HNO, int *
allowed);
```

Checks, whether the NodeObject is a job-node which accepts subnodes, like
JOB:REPEAT

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

allowed holds `1` if allowed and `0` otherwise after returning with `E_OK`

ret **E_OK** on success

E_FAILURE on failure

4.5.7.73. GetSelectEntriesCount

fct

```
int GetSelectEntriesCount(HController HC, HNodeObject HNO,
int *count);
```

If the NodeObject is of the type `VT_SEL` (`VAR:SELECT`) then this function gets the number of its entries. Use GetSelectEntry to retrieve an entry at index `n`.

> **TIP**
>
> To get the currently selected entry call GetNodeValueString.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

count holds the number of entries after returning with `E_OK`

ret **E_OK** on success

E_FAILURE on failure

see 4.5.7.74 GetSelectEntryLength, 4.5.7.75 GetSelectEntry,
4.5.7.3 GetNodeInfo, 4.5.7.82 GetNodeValueString

4.5.7.74. GetSelectEntryLength

fct

```
int GetSelectEntryLength(HController HC, HNodeObject HNO,
int index, int *length);
```

If the NodeObject is of type `VT_SEL` (`VAR:SELECT`) then this function gets the string length of the entry at `index`.

TIP

To get the currently selected entry call GetNodeValueString.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`index` is the entry number

`length` holds the `strlen` of the entry after returning with `E_OK`

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.7.82 GetNodeValueString, 4.5.7.75 GetSelectEntry, 4.5.7.73 GetSelectEntriesCount, 4.5.7.3 GetNodeInfo

4.5.7.75. GetSelectEntry

fct

```
int GetSelectEntry(HController HC, HNodeObject HNO, int
index, char *buffer, int bufferlen);
```

If the NodeObject is of type `VT_SEL` (`VAR:SELECT`) then this function gets the string of the entry at `index`.

TIP

To get the currently selected entry use GetNodeValueString.

Listing 4.23: GetSelectEntry example

```
int cnt;
if ( GetSelectEntriesCount(HC, HNO, &cnt) == E_OK ) {
  for (int i=0; i<cnt; ++i) {
    int buflen;
    if ( GetSelectEntryLength(HC, HNO, i, &buflen) ==
       E_OK ) {
      char *buffer = (char*)malloc(++buflen);
```

```
          if (GetSelectEntry(HC, HNO, i, buffer, buflen) ==
             E_OK) {
            printf(”Entry %i: %s\n”, i, buffer);
          }
          free(buffer);
        }
      }
    }
```

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

index is the entry number

buffer is the buffer for the entry

bufferlen is the length of the buffer

ret **E_OK** on success

E_NOSPACE if the buffer is too small

E_INVALID if the variable is not of type `VT_SEL`

E_FAILURE on failure

see 4.5.7.82 GetNodeValueString, 4.5.7.74 GetSelectEntryLength, 4.5.7.73 GetSelectEntriesCount, 4.5.7.3 GetNodeInfo

4.5.7.76. SetNodeValueBin

fct

```
int SetNodeValueBin(HController HC, HNodeObject HNO,const
unsigned char *buffer, int leninbytes);
```

If the NodeObject is of type `VT_BIN` (`VAR:BIN`) then this function sets a new value.

TIP

The buffer is copied so it can be freed immediately.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`buffer` is the pointer to the data

`leninbytes` is the length of the data in bytes

ret **`E_OK`** on success

`E_NALLOWED` if the NodeObject is not modifyable

`E_FAILURE` on failure

see 4.5.7.78 GetNodeValueBinCopy, 4.5.7.77 GetNodeValueBinCopyLen, 4.5.7.45 ModifyAllowed

4.5.7.77. GetNodeValueBinCopyLen

fct

```
int GetNodeValueBinCopyLen(HController HC, HNodeObject HNO);
```

If the NodeObject is of type `VT_BIN` (`VAR:BIN`) then this function gets the size in bytes.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

ret the size of the variable in bytes

`E_FAILURE` on failure

see 4.5.7.78 GetNodeValueBinCopy, 4.5.7.76 SetNodeValueBin

4.5.7.78. GetNodeValueBinCopy

fct

```
int GetNodeValueBinCopy(HController HC, HNodeObject HNO,
unsigned char *buffer, int len);
```

If the NodeObject is of type `VT_BIN` (`VAR:BIN`) then this function gets a copy of the data. The buffer has to have the size of `len` bytes.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`buffer` holds the buffer where the node value gets copied in after returning with `E_OK`

`len` is the size of the buffer in bytes

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.7.77 GetNodeValueBinCopyLen, 4.5.7.76 SetNodeValueBin

4.5.7.79. SetNodeValueBool

fct

```
int SetNodeValueBool(HController HC, HNodeObject HNO, int
flag);
```

If the NodeObject is of type `VT_BOOLEAN` (`VAR:BOOLEAN`) then this function sets its new value.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`flag` is the new value

ret **`E_OK`** on success

`E_NALLOWED` if the NodeObject is not modifyable

`E_INVALID` if the node is not of type `VT_BOOLEAN`

E_FAILURE on failure

see 4.5.7.80 GetNodeValueBool, 4.5.7.45 ModifyAllowed

4.5.7.80. GetNodeValueBool

fct

```
int GetNodeValueBool(HController HC, HNodeObject HNO, int *
flag);
```

If the NodeObject is of type `VT_BOOLEAN` (`VAR:BOOLEAN`) then this function gets its value.

> **TIP**
>
> With GetNodeInfo it is possible to get all information about a node with a single call.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

flag is the new value

ret **E_OK** on success

E_INVALID if the node is not of type `VT_BOOLEAN`

E_FAILURE on failure

see 4.5.7.79 SetNodeValueBool, 4.5.7.3 GetNodeInfo

4.5.7.81. GetNodeValueStringLen

fct

```
int GetNodeValueStringLen(HController HC, HNodeObject HNO);
```

Returns the `strlen` of the value.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

ret the `strlen`

`E_INVALID` if the node is not representable as a string

`E_FAILURE` on failure

see 4.5.7.82 GetNodeValueString, 4.5.7.3 GetNodeInfo

4.5.7.82. GetNodeValueString

fct

```
int GetNodeValueString(HController HC, HNodeObject HNO,
char *value, int maxlen);
```

Gets the string value of a NodeObject. This function can be called for NodeObjects of the following types:

- VT_STR (VAR:STRING)
- VT_TEXT (VAR:TEXT)
- VT_I32 (VAR:INT32)
- VT_I64 (VAR:INT64)
- VT_R32 (VAR:REAL32)
- VT_R64 (VAR:REAL64)
- VT_SEL (VAR:SELECT)
- VT_BOOLEAN (VAR:BOOLEAN)

If the function returns `E_OK` then the string value can be read from `value`. Call GetNodeValueStringLen to find out how big the buffer has to be.

Listing 4.24: GetNodeValueString example

```
int buflen = GetNodeValueStringLen(HC, HNO);
if (buflen > 0) {
  char *buffer = (char*)malloc(++buflen);
  if (GetNodeValueString(HC, HNO, buffer, buflen) ==
     E_OK) {
    printf(”Nodevalue: \%s\n”, buffer);
```

```
    }
    free(buffer);
  }
```

TIP

The decimal separator is a dot by default. To change this to your current locale call DisableFloatingPointToStringDot.
With GetNodeInfo it is possible to get all information about a node with a single call.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

`value` pointer to a buffer

`maxlen` size of the buffer

ret `E_OK` on success

`E_NOSPACE` if the buffer is too small

`E_INVALID` if the node is not representable as a string

`E_FAILURE` on failure

see 4.5.7.81 GetNodeValueStringLen, 4.5.7.87 SetNodeValueString, 4.5.7.3 GetNodeInfo

4.5.7.83. GetNodeValueInt32

fct

```
int GetNodeValueInt32(HController HC, HNodeObject HNO, int
*value);
```

Gets the int32-value of the NodeObject. Only valid if GetNodeType returned `VT_I32`. If the function returns `E_OK` then `value` has the current value.

> **TIP**
>
> With GetNodeInfo it is possible to get all information about a node with a single call.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`value` holds the value after the call returns with `E_OK`

ret **`E_OK`** on success

`E_INVALID` if the node is not a int32-node

`E_FAILURE` on failure

see 4.5.7.88 SetNodeValueInt32, 4.5.7.37 GetNodeType, 4.5.7.3 GetNodeInfo

4.5.7.84. GetNodeValueInt64

fct

```
int GetNodeValueInt64(HController HC, HNodeObject HNO,
int64 *value);
```

Gets the int64-value of a NodeObject. Only valid if GetNodeType returned `VT_I64`. If the function returns `E_OK` then `value` has the current value.

> **TIP**
>
> With GetNodeInfo it is possible to get all information about a node with a single call.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`value` holds the value after the call returns with `E_OK`

ret **`E_OK`** on success

`E_INVALID` if the node is not a int64-node

E_FAILURE on failure

see 4.5.7.89 SetNodeValueInt64, 4.5.7.37 GetNodeType, 4.5.7.3 GetNodeInfo

4.5.7.85. GetNodeValueReal32

fct

```
int GetNodeValueReal32(HController HC, HNodeObject HNO,
float *value);
```

Gets the real32-value of a NodeObject. Only valid if GetNodeType returned `VT_R32`. If the function returns `E_OK` then `value` has the current value.

> **TIP**
>
> With GetNodeInfo it is possible to get all information about a node with a single call.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

value holds the value after the call returns with `E_OK`

ret **E_OK** on success

E_INVALID if the node is not of type real32

E_FAILURE on failure

see 4.5.7.90 SetNodeValueReal32, 4.5.7.37 GetNodeType, 4.5.7.3 GetNodeInfo

4.5.7.86. GetNodeValueReal64

fct

```
int GetNodeValueReal64(HController HC, HNodeObject HNO,
double *value);
```

Gets the real64-value of a NodeObject. Only valid if GetNodeType returned `VT_R64`. If the function returns `E_OK` then **`value`** has the current value.

TIP

With GetNodeInfo it is possible to get all information about a node with a single call.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`value` holds the value after the call returns with `E_OK`

ret **`E_OK`** on success

`E_INVALID` if the node is not of type real64

`E_FAILURE` on failure

see 4.5.7.90 SetNodeValueReal32, 4.5.7.37 GetNodeType, 4.5.7.3 GetNodeInfo

4.5.7.87. SetNodeValueString

fct

```
int SetNodeValueString(HController HC, HNodeObject HNO,
const char *value);
```

Sets the string-value of the NodeObject if the NodeObject is of type `VT_STR`, `VT_TEXT`, `VT_I32`, `VT_I64`, `VT_R32`, `VT_R64`, `VT_SEL` or `VT_BOOLEAN`.

TIP

The decimal separator is a dot by default. To change this to your current locale call DisableFloatingPointToStringDot.
To apply the changes to the controller you have to call WriteNode.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`value` is the new string-value of the NodeObject

ret `E_OK` on success

`E_NALLOWED` if it is not allowed to modify the NodeObject

`E_FAILURE` on failure

see 4.5.1.6 DisableFloatingPointToStringDot 4.5.7.82 GetNodeValueString, 4.5.7.7 WriteNode, 4.5.7.45 ModifyAllowed

4.5.7.88. SetNodeValueInt32

fct

```
int SetNodeValueInt32(HController HC, HNodeObject HNO, int
value);
```

Sets the int32-value of the NodeObject.

> **TIP**
>
> To apply the changes to the controller you have to call WriteNode.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

`value` is the new int32-value of the NodeObject

ret `E_OK` on success

`E_NALLOWED` if it is not allowed to modify the NodeObject

`E_RANGE` if the value is less than the minimum value or greater than the maximum value

`E_FAILURE` on failure

see 4.5.7.83 GetNodeValueInt32, 4.5.7.7 WriteNode, 4.5.7.45 ModifyAllowed, 4.5.7.92 GetMin, 4.5.7.93 GetMax

4.5.7.89. SetNodeValueInt64

fct

```
int SetNodeValueInt64(HController HC, HNodeObject HNO,
int64 value);
```

Sets the int64-value of a NodeObject.

> **TIP**
>
> To apply the changes to the controller you have to call WriteNode.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`value` is the new int64-value of the NodeObject

ret **`E_OK`** on success

`E_NALLOWED` if it is not allowed to modify the NodeObject

`E_RANGE` if the value is less than the minimum value or greater than the maximum value

`E_FAILURE` on failure

see 4.5.7.84 GetNodeValueInt64, 4.5.7.7 WriteNode, 4.5.7.45 ModifyAllowed, 4.5.7.92 GetMin, 4.5.7.93 GetMax

4.5.7.90. SetNodeValueReal32

fct

```
int SetNodeValueReal32(HController HC, HNodeObject HNO,
float value);
```

Sets the real32-value of the NodeObject.

> **TIP**
>
> To apply the changes to the controller you have to call *WriteNode(HC, HNO)*.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`value` is the new real32-value of the NodeObject

ret **`E_OK`** on success

`E_NALLOWED` if it is not allowed to modify the NodeObject

`E_RANGE` if the value is less than the minimum value or greater than the maximum value

`E_FAILURE` on failure

see 4.5.7.85 GetNodeValueReal32, 4.5.7.7 WriteNode, 4.5.7.45 ModifyAllowed, 4.5.7.92 GetMin, 4.5.7.93 GetMax

4.5.7.91. SetNodeValueReal64

fct

```
int SetNodeValueReal64(HController HC, HNodeObject HNO,
double value);
```

Sets the real64-value of the NodeObject.

> **TIP**
>
> To apply the changes to the controller you have to call WriteNode.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`value` is the new real64-value of the NodeObject

ret **`E_OK`** on success

`E_NALLOWED` if it is not allowed to modify the NodeObject

`E_RANGE` if the value is less than the minimum value or greater than the maximum value

`E_FAILURE` on failure

see 4.5.7.86 GetNodeValueReal64, 4.5.7.7 WriteNode, 4.5.7.45 ModifyAllowed, 4.5.7.92 GetMin, 4.5.7.93 GetMax

4.5.7.92. GetMin

fct
```
int GetMin(HController HC, HNodeObject HNO, float *min);
```

Gets the minimum value of the NodeObject. This value is set by the controller and cannot be changed by the user. If the minimum value is greater than the maximum value then the value itself is unlimited.

> **TIP**
>
> With GetNodeInfo it is possible to get all information about a node with a single call.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`min` holds the minimum value of the NodeObject after returning with `E_OK`

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.7.93 GetMax, 4.5.7.3 GetNodeInfo

4.5.7.93. GetMax

fct
```
int GetMax(HController HC, HNodeObject HNO, float *max);
```

Gets the maximum value of the NodeObject. This value is set by the controller and cannot be changed by the user. If the minimum value is greater than the maximum value then the value itself is unlimited.

TIP

With GetNodeInfo it is possible to get all information about a node with a single call.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

`max` holds the maximum value of the NodeObject after returning with `E_OK`

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.7.92 GetMin, 4.5.7.3 GetNodeInfo

4.5.7.94. GetUnitLen

fct

```
int GetUnitLen(HController HC, HNodeObject HNO, int *len);
```

Gets the needed buffersize for the unit-string. Note, that the needed buffersize is without the trailing 0-byte.

TIP

With GetNodeInfo it is possible to get all information about a node with a single call.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

`len` holds the needed buffersize after returning with `E_OK`. Note, that the needed buffersize is without the trailing 0-byte

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.7.95 GetUnit, 4.5.7.3 GetNodeInfo

4.5.7.95. GetUnit

fct

```
int GetUnit(HController HC, HNodeObject HNO, char *unit,
int len);
```

Gets the unit for the given NodeObject as string.

Listing 4.25: GetUnit example

```
int len;
if ( GetUnitLen(HC, HNO, &len) == E_OK ) {
  if ( len > 0 ) {
    char *buf = (char*)malloc(sizeof(char)*(++len));
    if ( GetUnit(HC, HNO, buf, len) == E_OK ) {
      printf("Unit: %s\n", buf);
    }
    free(buf);
  }
}
```

TIP

With GetNodeInfo it is possible to get all information about a node with a single call.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

`unit` is the pointer to a buffer big enough for the unit

`len` is the buffersize

ret `E_OK` on success

`E_NOMEM` buffer too small

`E_FAILURE` on failure

see 4.5.7.94 GetUnitLen, 4.5.7.3 GetNodeInfo, 4.5.7.4 GetNodeInfoExt

4.5.7.96. GetSubnodesNamesLen

fct

```
int GetSubnodesNamesLen(HController HC, const char *
nodename, int *len);
```

Gets the strlen of all subnodes of the given node. To get the subnodes call GetSubnodesNames.

par **HC** is the handle to the controller

nodename is the node from which the subnodes are wanted

len holds the strlen of the job names after returning with `E_OK`

ret **E_OK** on success

E_FAILURE on failure

see 4.5.7.97 GetSubnodesNames

4.5.7.97. GetSubnodesNames

fct

```
int GetSubnodesNames(HController HC, const char *nodename,
char *buffer, int len);
```

Gets the names of all subnodes of the given node. The names of the subnodes are separated by semicolons (e.g. ”`usr.var.x;usr.var.y0`”).

Listing 4.26: GetSubnodesNames example

```
int buflen;
if ( GetSubnodesNamesLen(HC, HNO, &buflen) == E_OK ) {
  char *buffer = (char*)malloc(++buflen);
  if (GetSubnodesNames(HC, HNO, buffer, buflen) == E_OK)
      {
    printf(”NodeName: %s\n”, buffer);
  }
  free(buffer);
}
```

par **HC** is the handle to the controller

nodename is the node from which the subnodes are wanted

buffer is the pointer to the buffer for the subnode names

len is the size of the buffer

ret **E_OK** on success

E_NOSPACE if the buffer is too small

E_FAILURE on failure

see 4.5.7.96 GetSubnodesNamesLen

4.5.7.98. GetSubnodesHNOArray

fct

```
int GetSubnodesHNOArray(HController HC, HNodeObject
HNOparent, HNodeObject *buffer, int *count );
```

Returns all subnodes as HNodeObjects in a single call.

Listing 4.27: GetSubnodesHNOArray example

```
int subnodecount = GetSubnodesCount(HC, HNO);
HNodeObject *subnodes = (HNodeObject*)malloc(sizeof(
   HNodeObject)*subnodecount);
int ret = GetSubnodesHNOArray(HC, HNO, subnodes, &
   subnodecount);

if ( ret != E_OK ) {
  return -1;
}

for (int i=0; i< subnodecount; ++i) {
  ARG_NODEINFO *info = GetNodeInfo(HC, subnodes[i]);
  if ( info ) {
    printf("  %s\n", info->name);
    DestroyNodeInfo(info);
```

```
    }
}
```

par **HC** is the handle to the controller

HNOparent is the node of which the subnodes are wanted

buffer pointer to a buffer for the HNOs

count size of the buffer. On return it holds the actual number of elements

ret **E_OK** on success

E_NOSPACE if the buffer is too small

E_FAILURE on failure

see 4.5.7.16 GetSubnodesCount

4.5.7.99. IsNodeChildOf

fct

```
int IsNodeChildOf(HController HC, HNodeObject childHandle,
HNodeObject parentHandle);
```

Tests, whether a node is the child of another node or not.

Listing 4.28: IsNodeChildOf example

```
parentHandle = GetNode(HC, "usr.job");
childHandle = GetNode(HC, "usr.job.Job");
if ( IsNodeChildOf(HC, childHandle, parentHandle) ==
    E_OK ) {
  // usr.job.Job is a child of usr.job!
}
```

par **HC** is the handle to the controller

childHandle is the handle to the assumed child node

parentHandle is the handle to the assumed parent node

ret **E_OK** if childHandle is a child of parentHandle

E_INVALID if childHandle is not a child of parentHandle

E_FAILURE on failure

see 4.5.7.100 IsNodeDescendantOf

4.5.7.100. IsNodeDescendantOf

fct

```
int IsNodeDescendantOf(HController HC, HNodeObject
descendantHandle, HNodeObject parentHandle);
```

Tests, whether a node is the descendant of another node or not.

Listing 4.29: IsNodeDescendantOf example

```
parentHandle = GetNode(HC, "usr.job");
descendantHandle = GetNode(HC, "usr.job.Job.Shape");
if ( IsNodeDescendantOf(HC, descendantHandle,
   parentHandle) == E_OK ) {
  // usr.job.Job.Shape is a descendant of usr.job!
}
```

par **HC** is the handle to the controller

descendantHandle is the handle to the assumed descendant node

parentHandle is the handle to the assumed parent node

ret **E_OK** if descendantHandle is a descendant of parentHandle

E_INVALID if descendantHandle is not a descendant of parentHandle

E_FAILURE on failure

see 4.5.7.99 IsNodeChildOf

4.5.7.101. GetEditorHint

fct

```
int GetEditorHint(HController HC, HNodeObject HNO, char
hint[128]);
```

Gets the editor hint for a variable. This is only useful when SetAttributes was called before.

par **HC** is the handle to the controller

HNO handle the NodeObject

hint holds the editor hint after returning with `E_OK`

ret **E_OK** if descendantHandle is a descendant of parentHandle

E_UNIMP if the firmware does not support this feature

E_UNAVAIL if the SetAttributes was not called

E_FAILURE on failure

see 4.5.1.12 SetAttributes

4.5.8. NodeObjectCollections

Table of contents

4.5.8.1. CreateNodeObjectCollection 163
4.5.8.2. DestroyNodeObjectCollection 164
4.5.8.3. AddNodeObject 164
4.5.8.4. AddNodeObjectByName 165
4.5.8.5. AddNodeObjectSubtreeByName 165
4.5.8.6. GetSubnodes 166
4.5.8.7. RemoveNodeObject 167
4.5.8.8. GetNodeObjectCount 168
4.5.8.9. GetNodeObjectAtIndex 169
4.5.8.10. ReadNodeObjectCollection 169
4.5.8.11. WriteNodeObjectCollection 170
4.5.8.12. WriteNodeObjectCollectionSync 170

4.5.8.1. CreateNodeObjectCollection

fct

```
HNodeObjectCollection CreateNodeObjectCollection(
HController HC);
```

Creates a NodeObjectCollection. With NodeObjectCollections writing of more than 1 NodeObject simultaneously is usually faster.

par `HC` is the handle to the controller

ret handle to the collection

`ARG_INVALID_HANDLE_VALUE` if not successfull

see 4.5.8.2 DestroyNodeObjectCollection

4.5.8.2. DestroyNodeObjectCollection

fct

```
int DestroyNodeObjectCollection(HController HC,
HNodeObjectCollection HNOC);
```

Destroys a NodeObjectCollection. All NodeObjects in the Collection get also destroyed.

par **HC** is the handle to the controller

HNOC is the handle to the NodeObjectCollection

ret **E_OK** on success

E_FAILURE on failure

see 4.5.8.1 CreateNodeObjectCollection

4.5.8.3. AddNodeObject

fct

```
int AddNodeObject(HController HC, HNodeObjectCollection
HNOC, HNodeObject HNO);
```

Adds a NodeObject to a collection.

TIP

For performance reasons it is recommended to use AddNodeObjectByName to add NodeObjects to the NodeObjectCollection.

par **HC** is the handle to the controller

HNOC is the handle to the NodeObjectCollection

HNO is the handle to the NodeObject

ret **E_OK** on success

E_FAILURE on failure

see 4.5.8.4 AddNodeObjectByName,
4.5.8.5 AddNodeObjectSubtreeByName, 4.5.8.7 RemoveNodeObject

4.5.8.4. AddNodeObjectByName

fct

```
int AddNodeObjectByName(HController HC,
HNodeObjectCollection HNOC, const char *varname);
```

Adds a NodeObject to a collection.

par **`HC`** is the handle to the controller

`HNOC` is the handle to the NodeObjectCollection

`varname` is the complete path of the node

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.8.3 AddNodeObject, 4.5.8.5 AddNodeObjectSubtreeByName,
4.5.8.7 RemoveNodeObject

4.5.8.5. AddNodeObjectSubtreeByName

fct

```
int AddNodeObjectSubtreeByName(HController HC,
HNodeObjectCollection HNOC, const char *nodename, int level,
ValueChangeCallbackFunction callback);
```

Adds the subtree of the given node to NodeObjectCollection.

par **`HC`** is the handle to the controller

`HNOC` is the handle to the NodeObjectCollection

`nodename` is the name of the subtree

level defines how deep the tree structure shall be loaded where 1 is just on level and 9999 is the complete subtree , e.g. use `AddNodeObjectSubtreeByName(HC, HNOC, ”usr”, 9999, callback)` to get the complete `usr.*` subtree.

callback if this is not `NULL` then this callback will be attached to each node object

ret **E_OK** on success

E_FAILURE on failure

see 4.5.8.3 AddNodeObject, 4.5.8.4 AddNodeObjectByName, 4.5.8.7 RemoveNodeObject, 4.2.1.3 ValueChangeCallbackFunction

4.5.8.6. GetSubnodes

fct

```
int GetSubnodes(HController HC, HNodeObject HNO,
HNodeObjectCollection HNOC);
```

Fills `HNOC` with the children of `HNO`.

TIP

- This function only works correct if the variable cache is enabled with EnableVariableCache.
- The NodeObjectCollection should not contain any elements.

Listing 4.30: GetSubnodes example

```
// Print subnodes of usr.job
int cnt, len;
HNodeObjectCollection HNOC;
HNodeObject HNO;
char *str;

HNOC = CreateNodeObjectCollection(HC);
if ( HNOC == ARG_INVALID_HANDLE_VALUE ) { error(); }
HNO = GetNode(HC, ”usr.job”);
```

```
if ( HNO == ARG_INVALID_HANDLE_VALUE ) { error(); }
if ( GetSubnodes(HC, HNO, HNOC) != E_OK ) { error(); }
if ( GetNodeObjectCount(HC, HNOC, &cnt) != E_OK) { error
   (); }
for (int i=0; i<cnt; ++i) {
  if ( GetNodeObjectAtIndex(HC, HNOC, i, &HNO) != E_OK )
      { error(); }
  len = GetNodeNameLen(HC, HNO);
  if ( len < 1 ) { error(); }
  str = (char*)malloc(++len);
  if ( GetNodeName(HC, HNO, str, len) != E_OK ) { error
     (); }
  printf("%s", str);
  free(str);
}

DestroyNodeObjectCollection(HC, HNOC);
```

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

HNOC is the handle to the NodeObjectCollection

ret **E_OK** on success

E_FAILURE on failure

see 4.5.2.6 EnableVariableCache, 4.5.7.15 GetParentNode, 4.5.8.1 CreateNodeObjectCollection, 4.5.7.1 GetNode, 4.5.8.8 GetNodeObjectCount

4.5.8.7. RemoveNodeObject

fct

```
int RemoveNodeObject(HController HC, HNodeObjectCollection
HNOC, HNodeObject HNO);
```

Removes a NodeObject from a collection without destroying the NodeObject.

par `HC` is the handle to the controller

`HNOC` is the handle to the NodeObjectCollection

`HNO` is the handle to the NodeObject

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.8.3 AddNodeObject, 4.5.8.4 AddNodeObjectByName, 4.5.8.5 AddNodeObjectSubtreeByName

4.5.8.8. GetNodeObjectCount

fct

```
int GetNodeObjectCount(HController HC,
HNodeObjectCollection HNOC, int *cnt);
```

Gets the count of NodeObjects in the NodeObjectCollection.

par `HC` is the handle to the controller

`HNOC` is the handle to the NodeObjectCollection

`cnt` holds the number of NodeObjects after returning successfully

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.8.9 GetNodeObjectAtIndex

4.5.8.9. GetNodeObjectAtIndex

fct

```
int GetNodeObjectAtIndex(HController HC,
HNodeObjectCollection HNOC, int index, HNodeObject *HNO);
```

Gets the NodeObject at a given index in the NodeObjectCollection.

par `HC` is the handle to the controller

`HNOC` is the handle to the NodeObjectCollection

`index` is the index of the wanted NodeObject where the first item has index `0`

`HNO` is the handle to the NodeObject

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.8.8 GetNodeObjectCount

4.5.8.10. ReadNodeObjectCollection

fct

```
int ReadNodeObjectCollection(HController HC,
HNodeObjectCollection HNOC);
```

Reads a complete NodeObjectCollection from the Controller. This is only necessary if NodeObjects are added by calls to AddNodeObjectByName and the variable cache is disabled.

par `HC` is the handle to the controller

`HNOC` is the handle to the NodeObjectCollection

ret `E_OK` on success

`E_FAILURE` on failure

`E_NOEXIST` if a node in the NodeObjectCollection does not exist

see 4.5.8.4 AddNodeObjectByName, 4.5.8.11 WriteNodeObjectCollection, 4.5.8.12 WriteNodeObjectCollectionSync, 4.5.2.6 EnableVariableCache

4.5.8.11. WriteNodeObjectCollection

fct
```
int WriteNodeObjectCollection(HController HC,
HNodeObjectCollection HNOC);
```

Writes all NodeObjects in the collection in 1 step. This is much faster than writing each NodeObject in a seperate step. If changing these nodes forces other nodes to change then WriteNodeObjectCollectionSync should be used.

par `HC` is the handle to the controller

`HNOC` is the handle to the NodeObjectCollection

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.8.12 WriteNodeObjectCollectionSync

4.5.8.12. WriteNodeObjectCollectionSync

fct
```
int WriteNodeObjectCollectionSync(HController HC,
HNodeObjectCollection HNOC);
```

Writes all NodeObjects in the collection in 1 step. This is much faster than writing each NodeObject in a seperate step. This function waits until the firmware sents all updates. This is useful if setting a variable changes other variables.

par `HC` is the handle to the controller

`HNOC` is the handle to the NodeObjectCollection

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.8.11 WriteNodeObjectCollection

4.5.9. Jobs

Table of contents

4.5.9.1. GetJobInfo . . . 173
4.5.9.2. DestroyJobInfo . . . 174
4.5.9.3. GetJobNamesLen . . . 174
4.5.9.4. GetJobNames . . . 175
4.5.9.5. JobClearAll . . . 175
4.5.9.6. GetJobNodeTypesCount . . . 176
4.5.9.7. GetJobNodeTypeInfo . . . 177
4.5.9.8. DestroyJobNodeTypeInfo . . . 177
4.5.9.9. GetJobLines . . . 178
4.5.9.10. GetJobLinesAbort . . . 178
4.5.9.11. JobNodesOffset . . . 179
4.5.9.12. JobNodesScale . . . 179
4.5.9.13. JobNodesRotate . . . 180
4.5.9.14. JobNodesTransform . . . 180
4.5.9.15. GetTssInfo . . . 181
4.5.9.16. DestroyTssInfo . . . 182
4.5.9.17. RequestTssDataConnection . . . 183
4.5.9.18. CancelTssDataConnection . . . 183
4.5.9.19. BeginSpecialJob . . . 184
4.5.9.20. EndSpecialJob . . . 185

4.5.9.1. GetJobInfo

fct

```
ARG_JOBINFO* GetJobInfo(HController HC);
```

Gets information about jobs on the controller. It is guaranteed that this function always returns a valid pointer to a ARG_JOBINFO-structure. The returned information has to be destroyed with a call to DestroyJobInfo.

Listing 4.31: GetJobInfo example

```
int found = 0;
ARG_JOBINFO *jobinfo = GetJobInfo(HC);
if (jobinfo->jobcount == 0 ) {
  printf(”No jobs on the controller.\n”);
} else {
  if ( jobinfo->jobcount == 1)
    printf(”%i job on the controller:\n”, jobinfo->
       jobcount);
  else
    printf(”%i jobs on the controller:\n”, jobinfo->
       jobcount);

  for (int i=0; i < jobinfo->jobcount; ++i) {
    if ( strcmp(jobinfo->jobnames[i], jobinfo->
       selectednode) == 0 ) {
      printf(”\t*%i: %s\n”, i, jobinfo->jobnames[i]);
      found = 1;
    } else
      printf(”\t %i: %s\n”, i, jobinfo->jobnames[i]);
  }

  if ( !found && (strcmp(jobinfo->selectednode,””) != 0)
     )
    printf(”\t Other node selected: %s\n”, jobinfo->
       selectednode);
}

DestroyJobInfo(jobinfo);
```

par `HC` is the handle to the controller

ret pointer to a ARG_JOBINFO-structure with information about jobs on the controller

see 4.5.9.2 DestroyJobInfo, 4.1.2.3 ARG_JOBINFO

4.5.9.2. DestroyJobInfo

fct

```
void DestroyJobInfo(ARG_JOBINFO *info);
```

Destroys the ARG_JOBINFO-structure which was obtained by a call to GetJobInfo.

par `info` is the ARG_JOBINFO-structure which was obtained by GetJobInfo

see 4.5.9.1 GetJobInfo, 4.1.2.3 ARG_JOBINFO

4.5.9.3. GetJobNamesLen

fct

```
int GetJobNamesLen(HController HC, int *len);
```

> **⚠ NOTICE**
>
> This function is deprecated.
>
> - Use GetJobInfo instead.

Gets the strlen of all job names on the controller. To get the job names call GetJobNames.

par `HC` is the handle to the controller

`len` holds the strlen of the job names after returning succesfully

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.9.4 GetJobNames

4.5.9.4. GetJobNames

fct

```
int GetJobNames(HController HC, char *buffer, int len);
```

NOTICE

This function is deprecated.

- Use GetJobInfo instead.

Gets the names of the jobs loaded on the controller. The job names are seperated by semicolons; e.g. "`Job1;Job2`".

par **`HC`** is the handle to the controller

`buffer` is the pointer to the buffer for the job names

`len` is the size of the buffer

ret **`E_OK`** on success

`E_NOSPACE` if the buffer is too small

`E_FAILURE` on failure

see 4.5.9.3 GetJobNamesLen

4.5.9.5. JobClearAll

fct

```
int JobClearAll(HController HC);
```

Clears all Jobs from the controller.

par **`HC`** is the handle to the controller

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.6.1 LoadJob

4.5.9.6. GetJobNodeTypesCount

fct

```
int GetJobNodeTypesCount(HController HC, int *cnt);
```

Gets information about how many job node types are supported by the controller. Information about those types can be obtained with GetJobN-odeTypeInfo.

Listing 4.32: GetJobNodeTypesCount example

```
int cnt = 0;
ARG_JOBNODEINFO *jobnodeinfo = NULL;
if ( GetJobNodeTypesCount(HC, &cnt) != E_OK ) {
  printf("Failure getting the count of the JobNodeTypes\
     n");
} else {
  for ( int i=0; i<cnt; ++i) {
    jobnodeinfo = GetJobNodeTypeInfo(HC, i);
    if ( jobnodeinfo != NULL ) {
      printf("%s -\t\t ", jobnodeinfo->fullname);
      if ( jobnodeinfo->usercreatable )
        printf(" usercreatable");
      if ( jobnodeinfo->canmakelines )
        printf(" canmakelines");
      if ( jobnodeinfo->canpasslines )
        printf(" canpasslines");
      if ( jobnodeinfo->acceptssubnodes )
        printf(" acceptssubnodes");
      printf("\n");
    }
    DestroyJobNodeTypeInfo(jobnodeinfo);
  }
}
```

par `HC` is the handle to the controller

`cnt` holds the number of supported job node types after returning with `E_OK`

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.9.7 GetJobNodeTypeInfo, 4.5.9.8 DestroyJobNodeTypeInfo, 4.1.2.2 ARG_JOBNODEINFO

4.5.9.7. GetJobNodeTypeInfo

fct

```
ARG_JOBNODEINFO* GetJobNodeTypeInfo(HController HC, int
index);
```

Gets information about a specific job node type, see ARG_JOBNODEINFO. On error, `NULL` is returned. The obtained structure has to be destroyed with DestroyJobNodeTypeInfo. For a complete example see GetJobNodeTypesCount.

par `HC` is the handle to the controller

`index` is the index of the job node type

ret pointer to a info structure; `NULL` if not possible

see 4.5.9.7 GetJobNodeTypeInfo, 4.5.9.8 DestroyJobNodeTypeInfo, 4.1.2.2 ARG_JOBNODEINFO

4.5.9.8. DestroyJobNodeTypeInfo

fct

```
void DestroyJobNodeTypeInfo(ARG_JOBNODEINFO *info);
```

Destroys a job node type info structure which was obtained with GetJobNodeTypeInfo.

par `info` is the pointer to the job node type info structure (can be `NULL`)

see 4.5.9.7 GetJobNodeTypeInfo, 4.1.2.2 ARG_JOBNODEINFO

4.5.9.9. GetJobLines

fct

```
int GetJobLines(HController HC, HNodeObject HNO, int
doSubnodes, DWORD userHandle);
```

If calling GetJobLines then the firmware returns several packages of data. The JobLinesCallbackFunction is called for each package. After the last package callback has returned this function returns.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

`doSubnodes` If `1` then also line information of the subnodes will be retrieved

`userHandle` is the handle set by the user in ARG_LINEINFO

ret `E_OK` on success

`E_FAILURE` on failure

see 4.2.5.4 JobLinesCallbackFunction, 4.5.9.10 GetJobLinesAbort, 4.5.12.94 RegisterOnJobLines, 4.5.12.96 UnregisterOnJobLines

4.5.9.10. GetJobLinesAbort

fct

```
int GetJobLinesAbort(HController HC);
```

Aborts all calls to JobGetLines.

par `HC` is the handle to the controller

ret `E_OK` on success

`E_FAILURE` on failure

see 4.2.5.4 JobLinesCallbackFunction, 4.5.12.94 RegisterOnJobLines, 4.5.12.96 UnregisterOnJobLines

4.5.9.11. JobNodesOffset

fct

```
int JobNodesOffset(HController HC, HNodeObject HNO,
HNodeObjectCollection HNOC, float dX, float dY);
```

par **HC** ist the handle to the controller

HNO is the handle to the root node of the offset

HNOC is the handle to the NodeObjectCollection

float dX

float dY

ret **E_OK** on success

E_FAILURE on failure

see 4.5.9.12 JobNodesScale, 4.5.9.13 JobNodesRotate,
4.5.9.14 JobNodesTransform

4.5.9.12. JobNodesScale

fct

```
int JobNodesScale(HController HC, HNodeObject HNO,
HNodeObjectCollection HNOC, float x0, float y0, float sX,
float sY);
```

par **HC** is the handle to the controller

HNO is the handle to the root node of the scale

HNOC is the handle to the NodeObjectCollection

float x0

float y0

float sX

float sY

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.9.11 JobNodesOffset, 4.5.9.13 JobNodesRotate,
4.5.9.14 JobNodesTransform

4.5.9.13. JobNodesRotate

fct

```
int JobNodesRotate(HController HC, HNodeObject HNO,
HNodeObjectCollection HNOC, float cX, float cY, float dA);
```

par `HC` is the handle to the controller

`HNO` is the handle to the root node of the rotate

`HNOC` is the handle to the NodeObjectCollection

`float` cX

`float` cY

`float` dA

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.9.11 JobNodesOffset, 4.5.9.12 JobNodesScale,
4.5.9.14 JobNodesTransform

4.5.9.14. JobNodesTransform

fct

```
int JobNodesTransform(HController HC, HNodeObject HNO,
HNodeObjectCollection HNOC, float v00, float v01, float v02,
 float v10, float v11, float v12);
```

par `HC` is the handle to the controller

`HNO` is the handle to the root node of the transform

HNOC is the handle to a NodeObjectCollection

float v00

float v01

float v02

float v10

float v11

float v12

ret **E_OK** on success

E_FAILURE on failure

see 4.5.9.11 JobNodesOffset, 4.5.9.12 JobNodesScale, 4.5.9.13 JobNodesRotate

4.5.9.15. GetTssInfo

fct

```
ARG_TSS_INFO* GetTssInfo(HController HC);
```

Gets info for timed signal streams. The returned structure has to be freed using DestroyTssInfo.

Listing 4.33: GetTssInfo example

```
ARG_TSS_INFO *info = GetTssInfo(HC);
if ( info ) {
  if ( info->channelcount > 0 ) {
    for (int i=0; i<info->channelcount; ++i) {
      ARG_TSS_CHANNELTYPE *type = info->channeltype[i];
      if ( type ) {
        printf(”\nChanneltype %d\n”,(i+1));
        printf(”============\n”);
        printf(”  Name:                    %s\n”, type->
            name);
        printf(”  GetCoordinateSystemName: %s\n”, type->
            coordname);
        if ( type->axiscount == 0 ) {
```

```
                printf("  No Axisnames found\n");
              } else {
                for (int j=0; j<type->axiscount; ++j) {
                  printf("    Axis %2d: %s\n", j, type->
                     axisname[j]);
                }
              }
            }
          }
        } else {
          printf("No TimedSignalStream-Types found\n");
        }

        DestroyTssInfo(info);
      }
```

par `HC` is the handle to the controller

ret a pointer to the ARG_TSS_INFO on success, `NULL` otherwise

`E_FAILURE` on failure

see 4.5.9.16 DestroyTssInfo

4.5.9.16. DestroyTssInfo

fct

```
void DestroyTssInfo(ARG_TSS_INFO *info);
```

Frees a structure that was obtained by GetTssInfo. For an example see GetTssInfo.

par `HC` is the handle to the controller

ret a pointer to the ARG_TSS_INFO on success, `NULL` otherwise

`E_FAILURE` on failure

see 4.5.9.15 GetTssInfo, 4.5.9.17 RequestTssDataConnection, 4.5.9.18 CancelTssDataConnection

4.5.9.17. RequestTssDataConnection

fct

```
int RequestTssDataConnection(HController HC, const char *
channeltype, const char *specifier, int *id);
```

par `HC` is the handle to the controller

`channeltype` is the channel type (name field of ARG_TSS_CHANNELTYPE)

`specifier` depends on the channel type; `NULL` in most cases

`id` holds the connection `ID` after returning with `E_OK`. The connection `ID` has to be used to relate to incoming data and to close the connection

ret `E_OK` if the data connection could be established

`E_FAILURE` on failure

see 4.5.9.15 GetTssInfo, 4.5.9.18 CancelTssDataConnection

4.5.9.18. CancelTssDataConnection

fct

```
int CancelTssDataConnection(HController HC, int id);
```

par `HC` is the handle to the controller

`id` is the `ID` which was obtained by the call to RequestTssDataConnection

ret `E_OK` if the data connection can be cancelled

`E_FAILURE` on failure

see 4.5.9.15 GetTssInfo, 4.5.9.18 CancelTssDataConnection

4.5.9.19. BeginSpecialJob

fct

```
int BeginSpecialJob(HController HC, const char *name, int
verify, HNodeObject *HNO);
```

Creates a special job on the controller.

Listing 4.34: BeginSpecialJob example

```
HNodeObject HNO;
if ( BeginSpecialJob(HC, "Pincushion", 1, &HNO) == E_OK
   ) {
  assert(HNO != ARG_INVALID_HANDLE_VALUE);
  //...
  EndSpecialJob(HC);
}
```

TIP

This function is for internal use only.

par `HC` is the handle to the controller

`name` is the name of the special job

`verify` `1`, parameterize the job to output it with the valid scanfield correct, `0` otherwise

`HNO` holds the HNodeObject of the created job after returning with `E_OK`, `ARG_INVALID_HANDLE_VALUE` otherwise

ret `E_OK` if the special job has been created

`E_FAILURE` on failure

see 4.5.9.20 EndSpecialJob

4.5.9.20. EndSpecialJob

fct

```
int EndSpecialJob(HController HC);
```

TIP

This function is for internal use only.

par **HC** is the handle to the controller

ret **E_OK** If the last special job has been ended with this call

E_FAILURE on failure

see 4.5.9.19 BeginSpecialJob

4.5.10. Drivers

Table of contents

4.5.10.1. GetDriverInfo 186
4.5.10.2. DestroyDriverInfo 187

4.5.10.1. GetDriverInfo

fct

```
ARG_DRIVERINFO* GetDriverInfo(HController HC);
```

Gets information about the drivers which are available with the firmware in use on the controller.

Listing 4.35: GetDriverInfo example

```
ARG_DRIVERINFO *info = GetDriverInfo(HC);
if ( info != NULL ) {
  printf("Found %i drivers\n", info->drivercount);
  for (int i=0; i<info->drivercount; ++i) {
    printf("Driver: %s (Version: %s)\n", info->driver[i
       ]->name, info->driver[i]->version);
    printf("Vendor: %s\n", info->driver[i]->vendor);
    printf("Comment: %s\n", info->driver[i]->comment);
    printf("\n");
  }
  DestroyDriverInfo(info);
}
```

par `HC` is the handle to the controller

ret the pointer to the ARG_DRIVERINFO-structure with information about available drivers on the controller

see 4.5.10.2 DestroyDriverInfo, 4.1.2.5 ARG_DRIVERINFO, 4.1.2.6 ARG_SINGLEDRIVERINFO,

4.5.10.2. DestroyDriverInfo

fct
```
void DestroyDriverInfo(ARG_DRIVERINFO *info);
```

Destroys the structure obtained by GetDriverInfo.

par **`info`** the pointer to the ARG_DRIVERINFO-structure

see 4.5.10.1 GetDriverInfo

4.5.11. Devices

Table of contents

4.5.11.1. CreateDevice 189
4.5.11.2. GetDevice 189
4.5.11.3. DeleteDevice 190
4.5.11.4. DeleteDeviceByName 190
4.5.11.5. GetAllDeviceInfo 191
4.5.11.6. DestroyDeviceInfo 192
4.5.11.7. GetDeviceInfo 192
4.5.11.8. DestroySingleDeviceInfo 193
4.5.11.9. ActivateDevice 193
4.5.11.10. ActivateDeviceByName 194
4.5.11.11. DeactivateDevice 194
4.5.11.12. DeactivateDeviceByName 195
4.5.11.13. SetManaged 195
4.5.11.14. SetManagedByName 196
4.5.11.15. SetUnmanaged 196
4.5.11.16. SetUnmanagedByName 197
4.5.11.17. IsManaged 197
4.5.11.18. IsManagedByName 198
4.5.11.19. IsDeviceResettable 198
4.5.11.20. IsDeviceResettableByName 199
4.5.11.21. ResetDevice 199
4.5.11.22. ResetDeviceByName 200
4.5.11.23. CreatePenVar 201

4.5.11.1. CreateDevice

fct

```
int CreateDevice(HController HC, const char *drivername,
const char *devicename);
```

Creates a device from a driver.

Listing 4.36: CreateDevice example

```
if ( CreateDevice(HC, "IPG YLP", "My Laser") != E_OK ) {
  printf("Could not create device.");
}
```

par **HC** is the handle to the controller

drivername is the name of the driver

devicename is the name of the device. This name must be unique in the firmware

ret **E_OK** on success

E_FAILURE on failure

see 4.5.10.1 GetDriverInfo, 4.5.11.5 GetAllDeviceInfo, 4.5.11.9 ActivateDevice

4.5.11.2. GetDevice

fct

```
HDevice GetDevice(HController HC, const char *deviceName);
```

Returns the handle to the given device.

par **HC** is the handle to the controller

deviceName is the name of the device

ret the handle of the device

ARG_INVALID_HANDLE_VALUE if the device was not found

see 4.5.11.1 CreateDevice, 4.5.11.5 GetAllDeviceInfo

4.5.11.3. DeleteDevice

fct
```
int DeleteDevice(HController HC, HDevice HD);
```

Deletes the given device. The device is identified by its handle.

Listing 4.37: DeleteDevice example
```
HDevice HD = GetDevice(HC, "My Laser");
if ( HD != ARG_INVALID_HANDLE_VALUE ) {
  DeleteDevice(HC, HD);
}
```

par **HC** is the handle to the controller

HD is the handle to the device

ret **E_OK** on success

E_FAILURE on failure

see 4.5.11.4 DeleteDeviceByName, 4.5.11.1 CreateDevice, 4.5.11.5 GetAllDeviceInfo

4.5.11.4. DeleteDeviceByName

fct
```
int DeleteDeviceByName(HController HC, const char *
devicename);
```

Deletes the given device. The device is identified by its name.

Listing 4.38: DeleteDeviceByName example
```
DeleteDeviceByName(HC, "My Laser");
```

par **HC** is the handle to the controller

devicename is the name of the device

ret **E_OK** on success

E_FAILURE on failure

see 4.5.11.4 DeleteDeviceByName, 4.5.11.1 CreateDevice, 4.5.11.5 GetAllDeviceInfo

4.5.11.5. GetAllDeviceInfo

fct

```
ARG_DEVICEINFO* GetAllDeviceInfo(HController HC);
```

Returns a structure with information about all devices used on the controller. All strings in the structure are valid, so there is no need to test for `NULL`, but can contain emtpy strings ("").

Listing 4.39: GetAllDeviceInfo example

```
ARG_DEVICEINFO *info = GetAllDeviceInfo(HC);
if ( info ) {
  for (int i=0; i<info->devicecount; ++i) {
    ARG_SINGLEDEVICEINFO *singleinfo = device[i];
    printf("Device: %s\n");
  }
  DestroyDeviceInfo(info);
}
```

TIP

The returned structure has to be destroyed by using DestroyDeviceInfo.

par **HC** is the handle to the controller

ret the pointer to the ARG_DEVICEINFO-structure with information about the devices on the controller

see 4.5.11.6 DestroyDeviceInfo, 4.5.11.7 GetDeviceInfo, 4.1.2.8 ARG_DEVICEINFO, 4.1.2.9 ARG_SINGLEDEVICEINFO

4.5.11.6. DestroyDeviceInfo

fct
```
void DestroyDeviceInfo(ARG_DEVICEINFO *info);
```

Destroys the structure which was obtained by a call to GetAllDeviceInfo. All internally used memory will be destroyed.

par `info` is the pointer to the ARG_DEVICEINFO-structure

see 4.5.11.5 GetAllDeviceInfo

4.5.11.7. GetDeviceInfo

fct
```
ARG_SINGLEDEVICEINFO* GetDeviceInfo(HController HC, HDevice
 HD);
```

Returns information about a particular device on the controller. All strings in the structure are valid, so there is no need to test for `NULL`, but can contain emtpy strings (“”).

Listing 4.40: GetDeviceInfo example

```
HDevice HD = GetDevice(HC,”My Laser”);
if ( HD != ARG_INVALID_HANDLE_VALUE ) {
  ARG_SINGLEDEVICEINFO *info = GetDeviceInfo(HC, HD);
  if ( info ) {
    printf(”Devicename: %s\n”, info->name);
    DestroySingleDeviceInfo(info);
  }
}
```

TIP

The returned structure has to be destroyed by using DestroySingleDeviceInfo.

par `HC` is the handle to the controller

`HD` is the handle to the device

ret the pointer to the ARG_SINGLEDEVICEINFO-structure with information about the devices on the controller

see 4.5.11.8 DestroySingleDeviceInfo, 4.5.11.5 GetAllDeviceInfo, 4.1.2.8 ARG_DEVICEINFO, 4.1.2.9 ARG_SINGLEDEVICEINFO

4.5.11.8. DestroySingleDeviceInfo

fct
```
void DestroySingleDeviceInfo(ARG_SINGLEDEVICEINFO *info);
```

Destroys the structure obtained by GetDeviceInfo.

par **`info`** is the pointer to the ARG_SINGLEDEVICEINFO-structure

see 4.5.11.7 GetDeviceInfo

4.5.11.9. ActivateDevice

fct
```
int ActivateDevice(HController HC, HDevice HD);
```

Activates the given device. When this function returns, this does not mean that the device is active. The state of activation can be monitored by using RegisterOnDeviceStateChanged.

par **`HC`** is the handle to the controller

`HD` is the handle to the device

ret **`E_OK`** on success

`E_INVALID` if the device is already active

`E_FAILURE` on failure

see 4.5.12.86 RegisterOnDeviceStateChanged, 4.5.11.5 GetAllDeviceInfo, 4.5.11.7 GetDeviceInfo, 4.5.11.11 DeactivateDevice

4.5.11.10. ActivateDeviceByName

fct

```
int ActivateDeviceByName(HController HC, const char *
devicename);
```

Activates the given device. When this function returns, this does not mean that the device is active. The state of the activation can be monitored by using RegisterOnDeviceStateChanged.

par **HC** is the handle to the controller

devicename is the name of the device that shall be activated

ret **E_OK** on success

E_INVALID if the device is already active

E_FAILURE on failure

see 4.5.12.86 RegisterOnDeviceStateChanged, 4.5.11.5 GetAllDeviceInfo, 4.5.11.7 GetDeviceInfo, 4.5.11.11 DeactivateDevice

4.5.11.11. DeactivateDevice

fct

```
int DeactivateDevice(HController HC, HDevice HD);
```

Deactivates the given device. When this function returns, this does not mean that the device is inactive. The state of the deactivation can be monitored by using RegisterOnDeviceStateChanged.

par **HC** is the handle to the controller

HD is the handle to the device

ret **E_OK** on success

E_INVALID if the device is already inactive

E_FAILURE on failure

see 4.5.12.86 RegisterOnDeviceStateChanged, 4.5.11.5 GetAllDeviceInfo, 4.5.11.7 GetDeviceInfo, 4.5.11.9 ActivateDevice

4.5.11.12. DeactivateDeviceByName

fct

```
int DeactivateDeviceByName(HController HC, const char *
devicename);
```

Deactivates the given device. When this function returns, this does not mean that the device is inactive. The state of the deactivation can be monitored by using RegisterOnDeviceStateChanged.

par **`HC`** is the handle to the controller

`devicename` is the name of the device that shall be deactivated

ret **`E_OK`** on success

`E_INVALID` if the device is already inactive

`E_FAILURE` on failure

see 4.5.12.86 RegisterOnDeviceStateChanged, 4.5.11.5 GetAllDeviceInfo, 4.5.11.7 GetDeviceInfo, 4.5.11.9 ActivateDevice

4.5.11.13. SetManaged

fct

```
int SetManaged(HController HC, HDevice HD);
```

Sets the device to managed. The device is identified by its handle.

par **`HC`** is the handle to the controller

`HD` is the handle to the device

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.11.5 GetAllDeviceInfo, 4.5.11.7 GetDeviceInfo, 4.5.11.14 SetManagedByName, 4.5.11.15 SetUnmanaged, 4.5.11.16 SetUnmanagedByName, 4.5.11.17 IsManaged, 4.5.11.18 IsManagedByName

4.5.11.14. SetManagedByName

fct
```
int SetManagedByName(HController HC, const char *devicename
);
```

Sets the device to managed. The device is identified by its name.

par **HC** is the handle to the controller

devicename is the name of the device

ret **E_OK** on success

E_FAILURE on failure

see 4.5.11.5 GetAllDeviceInfo, 4.5.11.7 GetDeviceInfo, 4.5.11.13 SetManaged, 4.5.11.15 SetUnmanaged, 4.5.11.16 SetUnmanagedByName, 4.5.11.17 IsManaged, 4.5.11.18 IsManagedByName

4.5.11.15. SetUnmanaged

fct
```
int SetUnmanaged(HController HC, HDevice HD);
```

Sets the device to unmanaged. The device is identified by its handle.

par **HC** is the handle to the controller

HD is the handle to the device

ret **E_OK** on success

E_FAILURE on failure

see 4.5.11.5 GetAllDeviceInfo, 4.5.11.7 GetDeviceInfo, 4.5.11.16 SetUnmanagedByName, 4.5.11.13 SetManaged, 4.5.11.14 SetManagedByName, 4.5.11.17 IsManaged, 4.5.11.18 IsManagedByName

4.5.11.16. SetUnmanagedByName

fct

```
int SetUnmanagedByName(HController HC, const char *
devicename);
```

Set the device to unmanaged. The device is identified by its name.

par `HC` is the handle to the controller

`devicename` is the name of the device

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.11.5 GetAllDeviceInfo, 4.5.11.7 GetDeviceInfo, 4.5.11.15 SetUnmanaged, 4.5.11.13 SetManaged, 4.5.11.14 SetManagedByName, 4.5.11.17 IsManaged, 4.5.11.18 IsManagedByName

4.5.11.17. IsManaged

fct

```
int IsManaged(HController HC, HDevice HD, bool &managed);
```

Returns in variable `managed` whether the device is managed or not. The device is identified by its handle.

par `HC` is the handle to the controller

`HD` is the handle to the device

`managed` holds `true` if the device is managed, holds `false` otherwise, in case the function returns `E_OK`

ret `E_OK` on success

E_FAILURE on failure

see 4.5.11.5 GetAllDeviceInfo, 4.5.11.7 GetDeviceInfo, 4.5.11.18 IsManagedByName, 4.5.11.13 SetManaged, 4.5.11.14 SetManagedByName, 4.5.11.15 SetUnmanaged, 4.5.11.16 SetUnmanagedByName

4.5.11.18. IsManagedByName

fct

```
int IsManagedByName(HController HC, const char *devicename,
 bool &managed);
```

Returns in the variable **managed** whether the device is managed or not. The device is identified by its name.

par **HC** is the handle to the controller

devicename is the name of the device

managed If the function returns with E_OK: `true`: if the device is managed; `false` otherwise;

ret **E_OK** on success

E_FAILURE on failure

see 4.5.11.5 GetAllDeviceInfo, 4.5.11.7 GetDeviceInfo, 4.5.11.17 IsManaged, 4.5.11.13 SetManaged, 4.5.11.14 SetManagedByName, 4.5.11.15 SetUnmanaged, 4.5.11.16 SetUnmanagedByName

4.5.11.19. IsDeviceResettable

fct

```
int IsDeviceResettable(HController HC, HDevice HD);
```

Returns, whether it is possible to reset the device or not. The device is identified by its handle.

par **HC** is the handle to the controller

HD is the handle to the device

ret **E_OK** if the device can be reset

E_NALLOWED if the device can not be reset

E_FAILURE on failure

see 4.5.11.5 GetAllDeviceInfo, 4.5.11.7 GetDeviceInfo, 4.5.11.20 IsDeviceResettableByName, 4.5.11.21 ResetDevice

4.5.11.20. IsDeviceResettableByName

fct

```
int IsDeviceResettableByName(HController HC, const char *
devicename);
```

Returns, whether it is possible to reset the device or not. The device is identified by its name.

par **HC** is the handle to the controller

devicename is the name of the device

ret **E_OK** if the device can be reset

E_NALLOWED if the device can not be reset

E_FAILURE on failure

see 4.5.11.5 GetAllDeviceInfo, 4.5.11.7 GetDeviceInfo, 4.5.11.19 IsDeviceResettable, 4.5.11.22 ResetDeviceByName

4.5.11.21. ResetDevice

fct

```
int ResetDevice(HController HC, HDevice HD);
```

Resets the given device. The device is identified by its handle. When this function returns, this does not mean that the device has been reset. The state of the reset can be monitored by using RegisterOnDeviceStateChanged.

par **HC** is the handle to the controller

HD is the handle to the device

ret **E_OK** on success

E_NALLOWED if the device can not be reset

E_FAILURE on failure

see 4.5.12.86 RegisterOnDeviceStateChanged, 4.5.11.5 GetAllDeviceInfo, 4.5.11.7 GetDeviceInfo, 4.5.11.19 IsDeviceResettable, 4.5.11.22 ResetDeviceByName

4.5.11.22. ResetDeviceByName

fct

```
int ResetDeviceByName(HController HC, const char *
devicename);
```

Resets the given device. The device is identified by its name. When this function returns, this does not mean that the device has been reset. The state of the reset can be monitored by using RegisterOnDeviceStateChanged.

par **HC** is the handle to the controller

devicename is the name of the device

ret **E_OK** on success

E_NALLOWED if the device can not be reset

E_FAILURE on failure

see 4.5.12.86 RegisterOnDeviceStateChanged, 4.5.11.5 GetAllDeviceInfo, 4.5.11.7 GetDeviceInfo, 4.5.11.19 IsDeviceResettable, 4.5.11.21 ResetDevice

4.5.11.23. CreatePenVar

fct

```
int CreatePenVar(HController HC, const char *drivername,
const char *pennamevariable);
```

Creates a pen variable for a device.

par **`HC`** is the handle to the controller

`drivername` is the name of the driver; e.g. `LINEPAR`

`pennamevariable` is the path to the pen variable; e.g.
`usr.pens.MyPen.linepar.common.speed_m`

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.10.1 GetDriverInfo

4.5.12. Callbacks

Please see also 4.5.3 Error handling.

Table of contents

4.5.12.1. RegisterOnNodeModified . . . 205
4.5.12.2. UnregisterOnNodeModified . . . 206
4.5.12.3. UnregisterOnNodeModifiedSingle . . . 207
4.5.12.4. RegisterOnNodeModifiedGlobal . . . 207
4.5.12.5. UnregisterOnNodeModifiedGlobal . . . 208
4.5.12.6. RegisterOnValueChangedExt . . . 208
4.5.12.7. RegisterOnValueChangedGlobal . . . 209
4.5.12.8. UnregisterOnValueChangedGlobal . . . 210
4.5.12.9. UnregisterOnValueChangedSingleExt . . . 210
4.5.12.10. RegisterOnValueChanged . . . 211
4.5.12.11. UnregisterOnValueChanged . . . 212
4.5.12.12. UnregisterOnValueChangedSingle . . . 212
4.5.12.13. RegisterOnFlagsChangedExt . . . 213
4.5.12.14. UnregisterOnFlagsChangedSingleExt . . . 214
4.5.12.15. RegisterOnFlagsChangedGlobal . . . 214
4.5.12.16. UnregisterOnFlagsChangedGlobal . . . 215
4.5.12.17. RegisterOnFlagsChanged . . . 215
4.5.12.18. UnregisterOnFlagsChanged . . . 216
4.5.12.19. UnregisterOnFlagsChangedSingle . . . 216
4.5.12.20. RegisterOnNodeMoved . . . 217
4.5.12.21. UnregisterOnNodeMoved . . . 218
4.5.12.22. RegisterOnNodeMovedExt . . . 218
4.5.12.23. UnregisterOnNodeMovedSingle . . . 219
4.5.12.24. UnregisterOnNodeMovedSingleExt . . . 219
4.5.12.25. RegisterOnNodeCreatedExt . . . 220

4.5.12.26. UnregisterOnNodeCreatedSingleExt 220
4.5.12.27. RegisterOnNodeCreated 221
4.5.12.28. UnregisterOnNodeCreated 222
4.5.12.29. UnregisterOnNodeCreatedSingle 222
4.5.12.30. RegisterOnStartOfNodeCreatedRequest 223
4.5.12.31. UnregisterOnStartOfNodeCreatedRequest 223
4.5.12.32. RegisterOnEndOfNodeCreatedRequest 224
4.5.12.33. UnregisterOnEndOfNodeCreatedRequest 224
4.5.12.34. RegisterOnNodeDeletedExt 225
4.5.12.35. RegisterOnNodeDeletedGlobal 226
4.5.12.36. UnregisterOnNodeDeletedGlobal 227
4.5.12.37. UnregisterOnNodeDeletedGlobalSingle 227
4.5.12.38. UnregisterOnNodeDeletedSingleExt 228
4.5.12.39. RegisterOnNodeDeleted 228
4.5.12.40. UnregisterOnNodeDeleted 229
4.5.12.41. UnregisterOnNodeDeletedSingle 230
4.5.12.42. RegisterOnStartOfNodeDeletedRequest 230
4.5.12.43. UnregisterOnStartOfNodeDeletedRequest 231
4.5.12.44. RegisterOnEndOfNodeDeletedRequest 231
4.5.12.45. UnregisterOnEndOfNodeDeletedRequest 232
4.5.12.46. RegisterOnNodeWillDeletedGlobal 232
4.5.12.47. UnregisterOnNodeWillDeletedGlobal 233
4.5.12.48. RegisterOnNameChangedExt 234
4.5.12.49. UnregisterOnNameChangedSingleExt 235
4.5.12.50. RegisterOnNameChangedGlobal 236
4.5.12.51. UnregisterOnNameChangedGlobal 236
4.5.12.52. UnregisterOnNameChangedGlobalSingle 237
4.5.12.53. RegisterOnNameChanged 237
4.5.12.54. UnregisterOnNameChangedSingle 238
4.5.12.55. UnregisterOnNameChanged 238
4.5.12.56. RegisterOnNodeStateChanged 239

4.5.12.57. RegisterOnNodeStateChangedGlobal 240
4.5.12.58. UnregisterOnNodeStateChangedGlobal 240
4.5.12.59. UnregisterOnNodeStateChangedSingle 240
4.5.12.60. UnregisterOnNodeStateChanged 241
4.5.12.61. RegisterOnPLCChangedExt 242
4.5.12.62. UnregisterOnPLCChangedSingleExt 242
4.5.12.63. RegisterOnPLCChanged 243
4.5.12.64. UnregisterOnPLCChanged 244
4.5.12.65. UnregisterOnPLCChangedSingle 244
4.5.12.66. RegisterOnSysMsgExt 245
4.5.12.67. UnregisterOnSysMsgSingleExt 246
4.5.12.68. RegisterOnSysMsg 246
4.5.12.69. UnregisterOnSysMsg 247
4.5.12.70. UnregisterOnSysMsgSingle 248
4.5.12.71. RegisterOnSystemMessage 248
4.5.12.72. UnregisterOnSystemMessage 249
4.5.12.73. UnregisterOnSystemMessageSingle 249
4.5.12.74. RegisterOnDeviceCreated 250
4.5.12.75. UnregisterOnDeviceCreated 250
4.5.12.76. RegisterOnDeviceDeleted 251
4.5.12.77. UnregisterOnDeviceDeleted 251
4.5.12.78. RegisterOnDeviceActivated 252
4.5.12.79. UnregisterOnDeviceActivated 252
4.5.12.80. RegisterOnDeviceDeactivated 253
4.5.12.81. UnregisterOnDeviceDeactivated 253
4.5.12.82. RegisterOnDeviceDependencyAdded 254
4.5.12.83. UnregisterOnDeviceDependencyAdded 254
4.5.12.84. RegisterOnDeviceDependencyRemoved 255
4.5.12.85. UnregisterOnDeviceDependencyRemoved 255
4.5.12.86. RegisterOnDeviceStateChanged 256
4.5.12.87. UnregisterOnDeviceStateChanged 256

4.5.12.88. RegisterOnDeviceErrorStateChanged 257
4.5.12.89. UnregisterOnDeviceErrorStateChanged 257
4.5.12.90. RegisterOnDeviceParamStateChanged 258
4.5.12.91. UnregisterOnDeviceParamStateChanged 258
4.5.12.92. RegisterOnDevicePowerStateChanged 259
4.5.12.93. UnregisterOnDevicePowerStateChanged 259
4.5.12.94. RegisterOnJobLines 260
4.5.12.95. UnregisterOnJobLinesSingle 260
4.5.12.96. UnregisterOnJobLines 260
4.5.12.97. RegisterOnTssData 261
4.5.12.98. UnregisterOnTssDataSingle 261
4.5.12.99. UnregisterOnTssData 262
4.5.12.100. RegisterOnUpdateTssChannels 262
4.5.12.101. UnregisterOnUpdateTssChannelsSingle 262
4.5.12.102. UnregisterOnUpdateTssChannels 263

4.5.12.1. RegisterOnNodeModified

fct

```
int RegisterOnNodeModified(HController HC, const char *
varname, NodeModifiedCallbackFunction callback, void *
userpointer);
```

Registers a callback for a NodeModified-event. Each time the given variable changes its internal values (value, flags, index, ...) this function gets called. Please note, that this function is partially redundant with the following functions:
4.5.12.6 RegisterOnValueChangedExt,
4.5.12.10 RegisterOnValueChanged,
4.5.12.13 RegisterOnFlagsChangedExt,
4.5.12.17 RegisterOnFlagsChanged

TIP

It is possible to have more than 1 callback for each variable.

par **`HC`** is the handle to the controller

`varname` is the name of the variable; e.g. "`stat.time.TimeStr`"

`callback` is the callback function

`userpointer` this pointer is given as parameter in the callback function

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.2.1.1 NodeModifiedCallbackFunction,
4.5.12.2 UnregisterOnNodeModified,
4.5.12.3 UnregisterOnNodeModifiedSingle,
4.5.12.6 RegisterOnValueChangedExt,
4.5.12.10 RegisterOnValueChanged,
4.5.12.13 RegisterOnFlagsChangedExt,
4.5.12.17 RegisterOnFlagsChanged,
4.5.12.48 RegisterOnNameChangedExt,
4.5.12.53 RegisterOnNameChanged,
4.5.12.34 RegisterOnNodeDeletedExt, 4.5.12.39 RegisterOnNodeDeleted,
4.5.8.5 AddNodeObjectSubtreeByName

4.5.12.2. UnregisterOnNodeModified

fct

```
int UnregisterOnNodeModified(HController HC, const char *
varname);
```

Unregisters all callbacks of a NodeModified-event from the given variable.

par **`HC`** is the handle to the controller

`varname` is the name of the variable; e.g. "`stat.time.TimeStr`"

ret `E_OK` on success

`E_NOEXIST` if a callback for the given variable has not been registered

`E_FAILURE` on failure

see 4.5.12.1 RegisterOnNodeModified,
4.5.12.3 UnregisterOnNodeModifiedSingle

4.5.12.3. UnregisterOnNodeModifiedSingle

fct

```
int UnregisterOnNodeModifiedSingle(HController HC, const
char *varname, NodeModifiedCallbackFunction callback);
```

Unregisters the given callback of a NodeModified-event from the given variable.

par `HC` is the handle to the controller

`varname` is the name of the variable; e.g. “`stat.time.TimeStr`”

`callback` is the callback that shall be unregistered

ret `E_OK` on success

`E_NOEXIST` if the callback for that variable has not been registered

`E_FAILURE` on failure

see 4.5.12.2 UnregisterOnNodeModified, 4.5.12.1 RegisterOnNodeModified

4.5.12.4. RegisterOnNodeModifiedGlobal

fct

```
int RegisterOnNodeModifiedGlobal(HController HC,
NodeModifiedCallbackFunction callback, void *userpointer);
```

par `HC` is the handle to the controller

`callback` is the callback function

userpointer this pointer is given as parameter in the callback function

ret **E_OK** on success

E_FAILURE on failure

see 4.5.12.5 UnregisterOnNodeModifiedGlobal

4.5.12.5. UnregisterOnNodeModifiedGlobal

fct
```
int UnregisterOnNodeModifiedGlobal(HController HC);
```

par **HC** is the handle to the controller

ret **E_OK** on success

E_FAILURE on failure

see 4.5.12.4 RegisterOnNodeModifiedGlobal

4.5.12.6. RegisterOnValueChangedExt

fct
```
int RegisterOnValueChangedExt(HController HC, const char *
varname, ValueChangeCallbackFunctionExt callback, void *
userpointer);
```

Registers a callback for a ValueChange-event. Each time the given variable changes the callback gets called with a NodeObject holding the current value. After registering the callback is called initially with the current value of the node.

> **TIP**
>
> It is possible to have more than 1 callback for each variable.

par **HC** is the handle to the controller

varname is the name of the variable; e.g. "`stat.time.TimeStr`"

callback is the callback function

`userpointer` this pointer is given as parameter in the callback function

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.2.1.2 ValueChangeCallbackFunctionExt,
4.5.12.11 UnregisterOnValueChanged,
4.5.12.9 UnregisterOnValueChangedSingleExt,
4.5.12.10 RegisterOnValueChanged,
4.5.12.13 RegisterOnFlagsChangedExt,
4.5.12.48 RegisterOnNameChangedExt,
4.5.12.34 RegisterOnNodeDeletedExt,
4.5.8.5 AddNodeObjectSubtreeByName

4.5.12.7. RegisterOnValueChangedGlobal

fct

```
int RegisterOnValueChangedGlobal(HController HC,
ValueChangeCallbackFunctionExt callback, void *userpointer);
```

Registers a callback for a ValueChange-event. Each time a variable changes the callback gets called with a NodeObject holding the current value.

> **TIP**
>
> Only 1 global callback per controller is allowed at a time.

par **`HC`** is the handle to the controller

`callback` is the callback function

`userpointer` this pointer is given as parameter in the callback function

ret **`E_OK`** on success

`E_EXIST` if already 1 global ValueChange callback was registered

`E_FAILURE` on failure

see 4.2.1.2 ValueChangeCallbackFunctionExt,
4.5.12.8 UnregisterOnValueChangedGlobal,
4.5.12.11 UnregisterOnValueChanged,
4.5.12.9 UnregisterOnValueChangedSingleExt,
4.5.12.10 RegisterOnValueChanged,
4.5.12.13 RegisterOnFlagsChangedExt,
4.5.12.48 RegisterOnNameChangedExt,
4.5.12.34 RegisterOnNodeDeletedExt,
4.5.8.5 AddNodeObjectSubtreeByName

4.5.12.8. UnregisterOnValueChangedGlobal

fct

```
int UnregisterOnValueChangedGlobal(HController HC);
```

Unregisters the global callback of a ValueChange-event from the given controller.

par `HC` is the handle to the controller

ret `E_OK` on success

`E_NOEXIST` if a callback for that variable was not registered

`E_FAILURE` on failure

see 4.5.12.11 UnregisterOnValueChanged,
4.5.12.7 RegisterOnValueChangedGlobal

4.5.12.9. UnregisterOnValueChangedSingleExt

fct

```
int UnregisterOnValueChangedSingleExt(HController HC, const
 char *varname, ValueChangeCallbackFunctionExt callback);
```

Unregister a callback of a ValueChange-event from the given variable.

par `HC` is the handle to the controller

`varname` name of the variable; e.g. “`stat.time.TimeStr`”

callback the callback to be unregistered

ret **E_OK** on success

E_NOEXIST if the callback for that variable was not registered

E_FAILURE on failure

see 4.5.12.11 UnregisterOnValueChanged,
4.5.12.6 RegisterOnValueChangedExt

4.5.12.10. RegisterOnValueChanged

fct
```
int RegisterOnValueChanged(HController HC, const char *
varname, ValueChangeCallbackFunction callback);
```

Registers a callback for a ValueChange-event. Each time the given variable changes the callback gets called with a NodeObject holding the current value. After registration the callback is called initially with the current value of the node.

TIP

It is possible to have more than 1 callback for each variable.

par **HC** is the handle to the controller

varname is the name of the variable; e.g. "`stat.time.TimeStr`"

callback is the callback function

ret **E_OK** on success

E_FAILURE on failure

see 4.2.1.3 ValueChangeCallbackFunction,
4.5.12.6 RegisterOnValueChangedExt,
4.5.12.11 UnregisterOnValueChanged,
4.5.12.12 UnregisterOnValueChangedSingle,
4.5.12.17 RegisterOnFlagsChanged, 4.5.12.53 RegisterOnNameChanged,

4.5.12.39 RegisterOnNodeDeleted,
4.5.8.5 AddNodeObjectSubtreeByName

4.5.12.11. UnregisterOnValueChanged

fct

```
int UnregisterOnValueChanged(HController HC, const char *
varname);
```

Unregisters all callbacks of a ValueChange-event from the given variable.

par **`HC`** is the handle to the controller

`varname` is the name of the variable; e.g. "`stat.time.TimeStr`"

ret **`E_OK`** on success

`E_NOEXIST` if a callback for that variable has not been registered

`E_FAILURE` on failure

see 4.5.12.12 UnregisterOnValueChangedSingle,
4.5.12.10 RegisterOnValueChanged

4.5.12.12. UnregisterOnValueChangedSingle

fct

```
int UnregisterOnValueChangedSingle(HController HC, const
char *varname, ValueChangeCallbackFunction callback);
```

Unregisters a callback of a ValueChange-event from the given variable.

par **`HC`** is the handle to the controller

`varname` is the name of the variable; e.g. "`stat.time.TimeStr`"

`callback` is the callback that shall be unregistered

ret **`E_OK`** on success

`E_NOEXIST` if the callback for that variable has not been registered

E_FAILURE on failure

see 4.5.12.11 UnregisterOnValueChanged, 4.5.12.10 RegisterOnValueChanged

4.5.12.13. RegisterOnFlagsChangedExt

fct

```
int RegisterOnFlagsChangedExt(HController HC, const char *
varname, FlagsChangeCallbackFunctionExt callback, void *
userpointer);
```

Registers a callback for a FlagsChange-event. Each time the flags for the given variable change the callback gets called with a NodeObject holding the current value. After registration the callback is called initially with the current value of the flags.

> **TIP**
>
> It is possible to have more than 1 callback for each variable.

par **HC** is the handle to the controller

varname is the name of the variable; e.g. "`stat.time.TimeStr`"

callback is the callback function

userpointer this pointer is given as parameter in the callback function

ret **E_OK** on success

E_FAILURE on failure

see 4.2.1.4 FlagsChangeCallbackFunctionExt,
4.5.12.17 RegisterOnFlagsChanged,
4.5.12.6 RegisterOnValueChangedExt,
4.5.12.18 UnregisterOnFlagsChanged,
4.5.12.14 UnregisterOnFlagsChangedSingleExt

4.5.12.14. UnregisterOnFlagsChangedSingleExt

fct

```
int UnregisterOnFlagsChangedSingleExt(HController HC, const
 char *varname, FlagsChangeCallbackFunctionExt callback);
```

Unregisters a callback of a FlagsChange-event.

par **`HC`** is the handle to the controller

`varname` name of the variable; e.g. "`stat.time.TimeStr`"

`callback` is the callback function

ret **`E_OK`** on success

`E_NOEXIST` if the callback for that variable was not registered

`E_FAILURE` on failure

see 4.5.12.13 RegisterOnFlagsChangedExt,
4.5.12.17 RegisterOnFlagsChanged, 4.5.12.18 UnregisterOnFlagsChanged

4.5.12.15. RegisterOnFlagsChangedGlobal

fct

```
int RegisterOnFlagsChangedGlobal(HController HC,
FlagsChangeCallbackFunctionExt callback, void *userpointer);
```

Registers a callback of a FlagsChange-event.

par **`HC`** is the handle to the controller

`callback` is the callback function

`userpointer` this pointer is given as parameter in the callback function

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.12.16 UnregisterOnFlagsChangedGlobal

4.5.12.16. UnregisterOnFlagsChangedGlobal

fct

```
int UnregisterOnFlagsChangedGlobal(HController HC);
```

Unregisters a callback of a FlagsChange-event.

par **`HC`** is the handle to the controller

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.12.15 RegisterOnFlagsChangedGlobal

4.5.12.17. RegisterOnFlagsChanged

fct

```
int RegisterOnFlagsChanged(HController HC, const char *
varname,FlagsChangeCallbackFunction callback);
```

Registers a callback for a FlagsChange-event. Each time the flags for the given variable change, the callback gets called with a NodeObject holding the current value. After registration the callback is called initially with the current value of the flags.

> **TIP**
>
> It is possible to have more than 1 callback for each variable.

par **`HC`** is the handle to the controller

`varname` is the name of the variable; e.g. "`stat.time.TimeStr`"

`callback` is the the callback function

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.2.1.5 FlagsChangeCallbackFunction,
4.5.12.13 RegisterOnFlagsChangedExt,
4.5.12.10 RegisterOnValueChanged,
4.5.12.18 UnregisterOnFlagsChanged,
4.5.12.19 UnregisterOnFlagsChangedSingle

4.5.12.18. UnregisterOnFlagsChanged

fct

```
int UnregisterOnFlagsChanged(HController HC, const char *
varname);
```

Unregisters all callbacks of a FlagsChange-event.

par **`HC`** is the handle to the controller

`varname` is the name of the variable; e.g. "`stat.time.TimeStr`"

ret **`E_OK`** on success

`E_NOEXIST` if a callback for that variable has not been registered

`E_FAILURE` on failure

see 4.5.12.17 RegisterOnFlagsChanged,
4.5.12.13 RegisterOnFlagsChangedExt,
4.5.12.19 UnregisterOnFlagsChangedSingle

4.5.12.19. UnregisterOnFlagsChangedSingle

fct

```
int UnregisterOnFlagsChangedSingle(HController HC, const
char *varname, FlagsChangeCallbackFunction callback);
```

Unregisters a callback of a FlagsChange-event.

par **`HC`** is the handle to the controller

`varname` is the name of the variable; e.g. "`stat.time.TimeStr`"

`callback` is the callback function

ret **`E_OK`** on success

`E_NOEXIST` if the callback for that variable has not been registered

`E_FAILURE` on failure

see 4.5.12.17 RegisterOnFlagsChanged, 4.5.12.18 UnregisterOnFlagsChanged

4.5.12.20. RegisterOnNodeMoved

fct

```
int RegisterOnNodeMoved(HController HC,
NodeMovedCallbackFunction callback, void *userpointer);
```

Registers a callback for a NodeMoved-event. Each time a node is moved on the controller the callback function gets called.

> **TIP**
>
> It is possible to have more than 1 callback at a time.

par **`HC`** is the handle to the controller

`callback` is the callback that gets called when a node is moved on the controller

`userpointer` This pointer is passed as a parameter to the callback function

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.2.1.17 NodeMovedCallbackFunction,
4.5.12.21 UnregisterOnNodeMoved,
4.5.12.23 UnregisterOnNodeMovedSingle

4.5.12.21. UnregisterOnNodeMoved

fct

```
int UnregisterOnNodeMoved(HController HC);
```

Unregisters all callbacks of a NodeMoved-event.

par **`HC`** is the handle to the controller

ret **`E_OK`** on success

`E_NOEXIST` if a callback has not been registered

`E_FAILURE` on failure

see 4.5.12.20 RegisterOnNodeMoved,
4.5.12.23 UnregisterOnNodeMovedSingle

4.5.12.22. RegisterOnNodeMovedExt

fct

```
int RegisterOnNodeMovedExt(HController HC,
NodeMovedCallbackFunctionExt callback, void *userpointer);
```

Registers a callback for a NodeMoved-event. Each time a node is moved on the controller this callback function gets called.

> **TIP**
>
> It is possible to have more than 1 callback at a time.

par **`HC`** is the handle to the controller

`callback` is the callback that gets called when a node is moved on the controller

`userpointer` This pointer is passed as a parameter to the callback function

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.2.1.17 NodeMovedCallbackFunction,
4.5.12.21 UnregisterOnNodeMoved,
4.5.12.23 UnregisterOnNodeMovedSingle

4.5.12.23. UnregisterOnNodeMovedSingle

fct
```
int UnregisterOnNodeMovedSingle(HController HC,
NodeMovedCallbackFunction callback);
```

Unregisters a callback for a NodeMoved-event.

par **`HC`** is the handle to the controller

`callback` is the callback function

ret **`E_OK`** on success

`E_NOEXIST` if the callback has not been registered

`E_FAILURE` on failure

see 4.5.12.20 RegisterOnNodeMoved, 4.5.12.21 UnregisterOnNodeMoved

4.5.12.24. UnregisterOnNodeMovedSingleExt

fct
```
int UnregisterOnNodeMovedSingleExt(HController HC,
NodeMovedCallbackFunctionExt callback);
```

Unregisters a callback for a NodeMoved-event.

par **`HC`** is the handle to the controller

`callback` is the callback function

ret **`E_OK`** on success

`E_NOEXIST` if the callback has not been registered

`E_FAILURE` on failure

see 4.5.12.20 RegisterOnNodeMoved, 4.5.12.21 UnregisterOnNodeMoved

4.5.12.25. RegisterOnNodeCreatedExt

fct

```
int RegisterOnNodeCreatedExt(HController HC,
NodeCreatedCallbackFunctionExt callback, void *userpointer);
```

Registers a callback for NodeCreate-events. This callback gets called each time a new node has been created on the controller.

> **TIP**
>
> It is possible to have more than 1 callback at a time.

par **`HC`** is the handle to the controller

`callback` is the callback function

`userpointer` This pointer is passed as a parameter to the callback function

ret **`E_OK`** on success

`E_EXIST` if the callback has already been registered

`E_FAILURE` on failure

see 4.2.1.8 NodeCreatedCallbackFunctionExt,
4.5.12.27 RegisterOnNodeCreated, 4.5.12.28 UnregisterOnNodeCreated,
4.5.12.26 UnregisterOnNodeCreatedSingleExt

4.5.12.26. UnregisterOnNodeCreatedSingleExt

fct

```
int UnregisterOnNodeCreatedSingleExt(HController HC,
NodeCreatedCallbackFunctionExt callback);
```

Unregisters a callback for NodeCreate-events.

par **`HC`** is the handle to the controller

`callback` is the callback function

ret `E_OK` on success

`E_NOEXIST` if the callback has not been registered

`E_FAILURE` on failure

see 4.5.12.25 RegisterOnNodeCreatedExt,
4.5.12.28 UnregisterOnNodeCreated

4.5.12.27. RegisterOnNodeCreated

fct

```
int RegisterOnNodeCreated(HController HC,
NodeCreatedCallbackFunction callback);
```

Registers a callback for NodeCreate-events. This callback gets called each time a new node has been created on the controller.

> **TIP**
>
> It is possible to have more than 1 callback at a time.

par `HC` is the handle to the controller

`callback` is the callback function

ret `E_OK` on success

`E_EXIST` if the callback has already been registered

`E_FAILURE` on failure

see 4.2.1.9 NodeCreatedCallbackFunction,
4.5.12.28 UnregisterOnNodeCreated,
4.5.12.29 UnregisterOnNodeCreatedSingle

4.5.12.28. UnregisterOnNodeCreated

fct

```
int UnregisterOnNodeCreated(HController HC );
```

Unregisters all callback for NodeCreate-events.

par **`HC`** is the handle to the controller

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.12.27 RegisterOnNodeCreated,
4.5.12.29 UnregisterOnNodeCreatedSingle

4.5.12.29. UnregisterOnNodeCreatedSingle

fct

```
int UnregisterOnNodeCreatedSingle(HController HC,
NodeCreatedCallbackFunction callback);
```

Unregisters a callback for NodeCreate-events.

par **`HC`** is the handle to the controller

`callback` is the callback function

ret **`E_OK`** on success

`E_NOEXIST` if the callback has not been registered

`E_FAILURE` on failure

see 4.5.12.27 RegisterOnNodeCreated, 4.5.12.28 UnregisterOnNodeCreated

4.5.12.30. RegisterOnStartOfNodeCreatedRequest

fct

```
int RegisterOnStartOfNodeCreatedRequest(HController HC,
StartOfNodeCreatedRequestCallbackFunction callback, void *
userpointer);
```

Registers a callback for StartOfNodeCreate-events. This callback gets called each time a new set of NodeCreated requests is coming from the controller.

> **TIP**
>
> It is possible to have more than 1 callback at a time.

par `HC` is the handle to the controller

`callback` is the callback function

`userpointer` This pointer is passed as a parameter to the callback function

ret `E_OK` on success

`E_FAILURE` on failure

see 4.2.1.10 StartOfNodeCreatedRequestCallbackFunction,
4.5.12.32 RegisterOnEndOfNodeCreatedRequest

4.5.12.31. UnregisterOnStartOfNodeCreatedRequest

fct

```
int UnregisterOnStartOfNodeCreatedRequest(HController HC);
```

Unregisters a callback for StartOfNodeCreate-events.

par `HC` is the handle to the controller

ret `E_OK` on success

`E_FAILURE` on failure

see 4.2.1.10 StartOfNodeCreatedRequestCallbackFunction,
4.5.12.30 RegisterOnStartOfNodeCreatedRequest,

4.5.12.32. RegisterOnEndOfNodeCreatedRequest

fct

```
int RegisterOnEndOfNodeCreatedRequest(HController HC,
EndOfNodeCreatedRequestCallbackFunction callback, void *
userpointer);
```

Registers a callback for EndOfNodeCreate-events. This callback gets called each time a new set of NodeCreated requests is coming from the controller.

> **TIP**
>
> It is possible to have more than 1 callback at a time.

par **`HC`** is the handle to the controller

`callback` is the callback function

`userpointer` This pointer is passed as a parameter to the callback function

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.2.1.11 EndOfNodeCreatedRequestCallbackFunction,
4.5.12.32 RegisterOnEndOfNodeCreatedRequest

4.5.12.33. UnregisterOnEndOfNodeCreatedRequest

fct

```
int UnregisterOnEndOfNodeCreatedRequest(HController HC);
```

Unregisters a callback for EndOfNodeCreate-events.

par **`HC`** is the handle to the controller

ret `E_OK` on success

`E_FAILURE` on failure

see 4.2.1.11 EndOfNodeCreatedRequestCallbackFunction,
4.5.12.30 RegisterOnStartOfNodeCreatedRequest

4.5.12.34. RegisterOnNodeDeletedExt

fct

```
int RegisterOnNodeDeletedExt(HController HC, HNodeObject
HNO, NodeDeletedCallbackFunctionExt callback, void *
userpointer);
```

Registers a callback for NodeDelete-events. The callback gets called when the given NodeObject was deleted on the controller.

TIP

- In order to work correctly, also a ValueChangedCallback has to be registered for the same variable.
- It is possible to have more than 1 callback at a time.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

`callback` is the callback function

`userpointer` This pointer is passed as a parameter to the callback function

ret `E_OK` on success

`E_EXIST` if the callback has already been registered

`E_FAILURE` on failure

see 4.2.1.12 NodeDeletedCallbackFunctionExt,
4.5.12.39 RegisterOnNodeDeleted,
4.5.12.35 RegisterOnNodeDeletedGlobal,
4.5.12.40 UnregisterOnNodeDeleted,
4.5.12.38 UnregisterOnNodeDeletedSingleExt,
4.5.12.10 RegisterOnValueChanged

4.5.12.35. RegisterOnNodeDeletedGlobal

fct

```
int RegisterOnNodeDeletedGlobal(HController HC,
NodeDeletedCallbackFunctionExt callback, void *userpointer);
```

Registers a callback for all NodeDelete-events. The callback gets called when any NodeObject was deleted on the controller.

TIP

- This function works only correct, if the Variablecache was enabled with EnableVariableCache.
- Only 1 callback of this type can be registered.

par **`HC`** is the handle to the controller

`callback` is the callback function

`userpointer` This pointer is passed as a parameter to the callback function

ret **`E_OK`** on success

`E_EXIST` if the callback has already been registered

`E_FAILURE` on failure

see 4.2.1.12 NodeDeletedCallbackFunctionExt,
4.5.12.36 UnregisterOnNodeDeletedGlobal,
4.5.12.39 RegisterOnNodeDeleted, 4.5.12.40 UnregisterOnNodeDeleted,

4.5.12.38 UnregisterOnNodeDeletedSingleExt,
4.5.12.10 RegisterOnValueChanged

4.5.12.36. UnregisterOnNodeDeletedGlobal

fct

```
int UnregisterOnNodeDeletedGlobal(HController HC);
```

Unregisters a callback for the global NodeDelete-events.

par **`HC`** is the handle to the controller

ret **`E_OK`** on success

`E_NOEXIST` if a callback has not been registered

`E_FAILURE` on failure

see 4.5.12.34 RegisterOnNodeDeletedExt,
4.5.12.40 UnregisterOnNodeDeleted,
4.5.12.35 RegisterOnNodeDeletedGlobal

4.5.12.37. UnregisterOnNodeDeletedGlobalSingle

fct

```
int UnregisterOnNodeDeletedGlobalSingle(HController HC,
NodeDeletedCallbackFunctionExt callback);
```

Unregisters a callback for the global NodeDelete-events.

par **`HC`** is the handle to the controller

`callback` is the callback function that shall be unregistered

ret **`E_OK`** on success

`E_NOEXIST` if the callback has not been registered

`E_FAILURE` on failure

see 4.5.12.34 RegisterOnNodeDeletedExt,
4.5.12.40 UnregisterOnNodeDeleted,
4.5.12.35 RegisterOnNodeDeletedGlobal

4.5.12.38. UnregisterOnNodeDeletedSingleExt

fct

```
int UnregisterOnNodeDeletedSingleExt(HController HC,
HNodeObject HNO, NodeDeletedCallbackFunctionExt callback);
```

Unregisters a callback for NodeDelete-events.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`callback` is the the callback function

ret **`E_OK`** on success

`E_NOEXIST` if the callback has not been registered

`E_FAILURE` on failure

see 4.5.12.34 RegisterOnNodeDeletedExt,
4.5.12.40 UnregisterOnNodeDeleted

4.5.12.39. RegisterOnNodeDeleted

fct

```
int RegisterOnNodeDeleted(HController HC, HNodeObject HNO,
NodeDeletedCallbackFunction callback);
```

Registers a callback for NodeDelete-events. The callback gets called when the given NodeObject was deleted on the controller.

TIP

- In order to work correctly, also a ValueChangedCallback has to be registered for the same variable.
- It is possible to have more than 1 callback at a time.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

`callback` is the callback function

ret `E_OK` on success

`E_EXIST` if a callback of this type has already been registered

`E_FAILURE` on failure

see 4.2.1.13 NodeDeletedCallbackFunction,
4.5.12.40 UnregisterOnNodeDeleted,
4.5.12.41 UnregisterOnNodeDeletedSingle,
4.5.12.10 RegisterOnValueChanged

4.5.12.40. UnregisterOnNodeDeleted

fct

```
int UnregisterOnNodeDeleted(HController HC, HNodeObject HNO
);
```

Unregisters all callbacks for NodeDelete-events.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.12.39 RegisterOnNodeDeleted,
4.5.12.41 UnregisterOnNodeDeletedSingle

4.5.12.41. UnregisterOnNodeDeletedSingle

fct

```
int UnregisterOnNodeDeletedSingle(HController HC,
HNodeObject HNO, NodeDeletedCallbackFunction callback );
```

Unregisters a callback for NodeDelete-events.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

callback is the callback function

ret **E_OK** on success

E_NOEXIST if a callback of this type has not been registered

E_FAILURE on failure

see 4.5.12.39 RegisterOnNodeDeleted, 4.5.12.40 UnregisterOnNodeDeleted

4.5.12.42. RegisterOnStartOfNodeDeletedRequest

fct

```
int RegisterOnStartOfNodeDeletedRequest(HController HC,
StartOfNodeDeletedRequestCallbackFunction callback, void *
userpointer);
```

Registers a callback for StartOfNodeDeleted-events. The callback gets called each time a new set of NodeDeleted requests is coming from the controller.

> **TIP**
>
> It is possible to have more than 1 callback at a time.

par **HC** is the handle to the controller

callback is the callback function

userpointer This pointer is passed as a parameter to the callback function

ret `E_OK` on success

`E_FAILURE` on failure

see 4.2.1.14 StartOfNodeDeletedRequestCallbackFunction,
4.5.12.44 RegisterOnEndOfNodeDeletedRequest

4.5.12.43. UnregisterOnStartOfNodeDeletedRequest

fct

```
int UnregisterOnStartOfNodeDeletedRequest(HController HC);
```

Unregisters a callback for StartOfNodeDeleted-events.

par `HC` is the handle to the controller

ret `E_OK` on success

`E_FAILURE` on failure

see 4.2.1.14 StartOfNodeDeletedRequestCallbackFunction,
4.5.12.42 RegisterOnStartOfNodeDeletedRequest

4.5.12.44. RegisterOnEndOfNodeDeletedRequest

fct

```
int RegisterOnEndOfNodeDeletedRequest(HController HC,
EndOfNodeDeletedRequestCallbackFunction callback, void *
userpointer);
```

Registers a callback for EndOfNodeDeleted-events. The callback gets called each time a new set of NodeDeleted-requests is coming from the controller.

> **TIP**
>
> It is possible to have more than 1 callback at a time.

par `HC` is the handle to the controller

`callback` is the callback function

userpointer This pointer is passed as a parameter to the callback function

ret **E_OK** on success

E_FAILURE on failure

see 4.2.1.15 EndOfNodeDeletedRequestCallbackFunction, 4.5.12.44 RegisterOnEndOfNodeDeletedRequest

4.5.12.45. UnregisterOnEndOfNodeDeletedRequest

fct

```
int UnregisterOnEndOfNodeDeletedRequest(HController HC);
```

Unregisters a callback for EndOfNodeDeleted-events.

par **HC** is the handle to the controller

ret **E_OK** on success

E_FAILURE on failure

see 4.2.1.15 EndOfNodeDeletedRequestCallbackFunction, 4.5.12.42 RegisterOnStartOfNodeDeletedRequest

4.5.12.46. RegisterOnNodeWillDeletedGlobal

fct

```
int RegisterOnNodeWillDeletedGlobal(HController HC,
NodeWillBeDeletedCallbackFunction callback, void *
userpointer);
```

Registers a callback for all NodeDelete-events. The callback gets called when any NodeObject was deleted on the controller.

> **TIP**
>
> - This function works only correct, if the Variablecache was enabled with EnableVariableCache.
> - Only 1 callback of this type can be registered.

par **`HC`** is the handle to the controller

`callback` is the callback function

`userpointer` This pointer is passed as a parameter to the callback function

ret **`E_OK`** on success

`E_EXIST` if the callback has already been registered

`E_FAILURE` on failure

see 4.2.1.12 NodeDeletedCallbackFunctionExt,
4.5.12.36 UnregisterOnNodeDeletedGlobal,
4.5.12.39 RegisterOnNodeDeleted, 4.5.12.40 UnregisterOnNodeDeleted,
4.5.12.38 UnregisterOnNodeDeletedSingleExt,
4.5.12.10 RegisterOnValueChanged

4.5.12.47. UnregisterOnNodeWillDeletedGlobal

fct

```
int UnregisterOnNodeWillDeletedGlobal(HController HC);
```

Registers a callback for all NodeWillBeDelete-events. The callback gets called when any NodeObject will be deleted on the controller.

> **TIP**
>
> - This function works only correct, if the Variablecache was enabled with EnableVariableCache.
> - Only 1 callback of this type can be registered.

par **`HC`** is the handle to the controller

callback is the callback function

userpointer This pointer is passed as a parameter to the callback function

ret **E_OK** on success

E_EXIST if the callback has already been registered

E_FAILURE on failure

see 4.2.1.12 NodeDeletedCallbackFunctionExt,
4.5.12.36 UnregisterOnNodeDeletedGlobal,
4.5.12.39 RegisterOnNodeDeleted, 4.5.12.40 UnregisterOnNodeDeleted,
4.5.12.38 UnregisterOnNodeDeletedSingleExt,
4.5.12.10 RegisterOnValueChanged

4.5.12.48. RegisterOnNameChangedExt

fct

```
int RegisterOnNameChangedExt(HController HC, HNodeObject
HNO, NameChangeCallbackFunctionExt callback, void *
userpointer);
```

Registers a callback for NameChange-events. The callback gets called when a NodeObject changes its name.

TIP

- It is possible to have more than 1 callback for each variable at a time.
- In order to work correctly, also a ValueChangedCallback has to be registered for the same variable.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

callback is the callback function

userpointer This pointer is passed as a parameter to the callback function

ret **E_OK** on success

E_EXIST if the callback has already been registered

E_FAILURE on failure

see 4.2.1.6 NameChangeCallbackFunctionExt,
4.5.12.53 RegisterOnNameChanged,
4.5.12.55 UnregisterOnNameChanged,
4.5.12.49 UnregisterOnNameChangedSingleExt,
4.5.12.10 RegisterOnValueChanged

4.5.12.49. UnregisterOnNameChangedSingleExt

fct

```
int UnregisterOnNameChangedSingleExt(HController HC,
HNodeObject HNO, NameChangeCallbackFunctionExt callback);
```

Unregisters a callback for NameChange-events for the given NodeObject.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

callback is the callback function

ret **E_OK** on success

E_NOEXIST if the callback has not been registered

E_FAILURE on failure

see 4.5.12.55 UnregisterOnNameChanged,
4.5.12.48 RegisterOnNameChangedExt

4.5.12.50. RegisterOnNameChangedGlobal

fct

```
int RegisterOnNameChangedGlobal(HController HC,
NameChangeCallbackFunctionExt callback, void *userpointer);
```

Registers a callback for NameChange-events.

par **`HC`** is the handle to the controller

`callback` is the callback function

`userpointer` This pointer is given as parameter in the callback function

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.12.51 UnregisterOnNameChangedGlobal,
4.5.12.52 UnregisterOnNameChangedGlobalSingle

4.5.12.51. UnregisterOnNameChangedGlobal

fct

```
int UnregisterOnNameChangedGlobal(HController HC);
```

Unregisters a callback for NameChange-events.

par **`HC`** is the handle to the controller

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.12.50 RegisterOnNameChangedGlobal,
4.5.12.52 UnregisterOnNameChangedGlobalSingle

4.5.12.52. UnregisterOnNameChangedGlobalSingle

fct

```
int UnregisterOnNameChangedGlobalSingle(HController HC,
NameChangeCallbackFunctionExt callback);
```

par **`HC`** is the handle to the controller

`callback` is the callback function

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.12.50 RegisterOnNameChangedGlobal,
4.5.12.51 UnregisterOnNameChangedGlobal

4.5.12.53. RegisterOnNameChanged

fct

```
int RegisterOnNameChanged(HController HC, HNodeObject HNO,
NameChangeCallbackFunction callback);
```

Registers a callback for NameChange-events. The callback gets called when a NodeObject changes its name.

> **TIP**
>
> It is possible to have more than 1 callback for each variable at a time. In order to work correctly, also a ValueChangedCallback has to be registered for the same variable.

par **`HC`** is the handle to the controller

`HNO` is the handle to the NodeObject

`callback` is the callback function

ret **`E_OK`** on success

`E_EXIST` if the callback has already been registered

`E_FAILURE` on failure

see 4.2.1.7 NameChangeCallbackFunction,
4.5.12.55 UnregisterOnNameChanged,
4.5.12.54 UnregisterOnNameChangedSingle,
4.5.12.10 RegisterOnValueChanged

4.5.12.54. UnregisterOnNameChangedSingle

fct

```
int UnregisterOnNameChangedSingle(HController HC,
HNodeObject HNO, NameChangeCallbackFunction callback);
```

Unregisters a callback for NameChange-events for the given NodeObject.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

callback is the callback function

ret **E_OK** on success

E_NOEXIST if the callback has not been registered

E_FAILURE on failure

see 4.5.12.55 UnregisterOnNameChanged,
4.5.12.53 RegisterOnNameChanged

4.5.12.55. UnregisterOnNameChanged

fct

```
int UnregisterOnNameChanged(HController HC, HNodeObject HNO
);
```

Unregisters all callbacks for NameChange-events for the given NodeObject.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

ret `E_OK` on success

`E_NOEXIST` if a callback has not been registered

`E_FAILURE` on failure

see 4.5.12.54 UnregisterOnNameChangedSingle,
4.5.12.53 RegisterOnNameChanged

4.5.12.56. RegisterOnNodeStateChanged

fct

```
int RegisterOnNodeStateChanged(HController HC, HNodeObject
HNO, NodeStateChangeCallbackFunction callback, void *
userpointer);
```

Registers a callback for NodeStateChange-events. The callback gets called when a NodeObject changes its state.

> **TIP**
>
> It is possible to have more than 1 callback for each variable.

par `HC` is the handle to the controller

`HNO` is the handle to the NodeObject

`callback` is the callback function

`userpointer` This pointer is passed as a parameter to the callback function

ret `E_OK` on success

`E_EXIST` if the callback has already been registered

`E_FAILURE` on failure

see 4.2.1.19 NodeStateChangeCallbackFunction,
4.5.12.60 UnregisterOnNodeStateChanged,
4.5.12.59 UnregisterOnNodeStateChangedSingle

4.5.12.57. RegisterOnNodeStateChangedGlobal

fct

```
int RegisterOnNodeStateChangedGlobal(HController HC,
NodeStateChangeCallbackFunction callback, void *userpointer
);
```

par **HC** is the handle to the controller

callback is the callback function

userpointer This pointer is given as parameter in the callback function

ret **E_OK** on success

E_FAILURE on failure

see 4.5.12.58 UnregisterOnNodeStateChangedGlobal

4.5.12.58. UnregisterOnNodeStateChangedGlobal

fct

```
int UnregisterOnNodeStateChangedGlobal(HController HC);
```

par **HC** is the handle to the controller

ret **E_OK** on success

E_FAILURE on failure

see 4.5.12.57 RegisterOnNodeStateChangedGlobal

4.5.12.59. UnregisterOnNodeStateChangedSingle

fct

```
int UnregisterOnNodeStateChangedSingle(HController HC,
HNodeObject HNO, NodeStateChangeCallbackFunction callback);
```

Unregisters a callback for NodeStateChange-events for the given NodeObject.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

callback is the callback function

ret **E_OK** on success

E_NOEXIST if the callback has not been registered

E_FAILURE on failure

see 4.5.12.60 UnregisterOnNodeStateChanged,
4.5.12.56 RegisterOnNodeStateChanged

4.5.12.60. UnregisterOnNodeStateChanged

fct

```
int UnregisterOnNodeStateChanged(HController HC,
HNodeObject HNO);
```

Unregisters all callbacks for NodeStateChanged-events for the given NodeObject.

par **HC** is the handle to the controller

HNO is the handle to the NodeObject

ret **E_OK** on success

E_NOEXIST if a callback has not been registered

E_FAILURE on failure

see 4.5.12.59 UnregisterOnNodeStateChangedSingle,
4.5.12.56 RegisterOnNodeStateChanged

4.5.12.61. RegisterOnPLCChangedExt

fct

```
int RegisterOnPLCChangedExt(HController HC,
PLCChangeCallbackFunctionExt callback, void *userpointer);
```

Registers a callback for a PLCChange-event. Each time the PLC state changes on the controller the callback function gets called. After registration the callback is being called initially with the current PLC state.

TIP

It is possible to have more than 1 callback at a time.

par **HC** is the handle to the controller

callback is the callback function

userpointer This pointer is passed as a parameter to the callback function

ret **E_OK** on success

E_EXIST if the callback has already been registered

E_FAILURE on failure

see 4.2.2.1 PLCChangeCallbackFunctionExt,
4.5.12.63 RegisterOnPLCChanged, 4.5.12.64 UnregisterOnPLCChanged,
4.5.12.65 UnregisterOnPLCChangedSingle

4.5.12.62. UnregisterOnPLCChangedSingleExt

fct

```
int UnregisterOnPLCChangedSingleExt(HController HC,
PLCChangeCallbackFunctionExt callback);
```

Unregisters a callback for a PLCChange-event.

par **HC** is the handle to the controller

callback is the callback function

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.12.61 RegisterOnPLCChangedExt,
4.5.12.64 UnregisterOnPLCChanged

4.5.12.63. RegisterOnPLCChanged

fct

```
int RegisterOnPLCChanged(HController HC,
PLCChangeCallbackFunction callback);
```

Registers a callback for a PLCChange-event. Each time the PLC state changes on the controller the callback function gets called. After registation the callback is being called initially with the current PLC state.

TIP

It is possible to have more than 1 callback at a time.

par **`HC`** is the handle to the controller

`callback` is the callback function

ret **`E_OK`** on success

`E_EXIST` if the callback has already been registered

`E_FAILURE` on failure

see 4.2.2.2 PLCChangeCallbackFunction,
4.5.12.64 UnregisterOnPLCChanged,
4.5.12.65 UnregisterOnPLCChangedSingle

4.5.12.64. UnregisterOnPLCChanged

fct

```
int UnregisterOnPLCChanged(HController HC);
```

Unregisters all callbacks for a PLC-event.

par **`HC`** is the handle to the controller

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.12.63 RegisterOnPLCChanged,
4.5.12.65 UnregisterOnPLCChangedSingle

4.5.12.65. UnregisterOnPLCChangedSingle

fct

```
int UnregisterOnPLCChangedSingle(HController HC,
PLCChangeCallbackFunction callback);
```

Unregisters a callback for a PLC-event.

par **`HC`** is the handle to the controller

`callback` is the callback function

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.12.63 RegisterOnPLCChanged, 4.5.12.64 UnregisterOnPLCChanged

4.5.12.66. RegisterOnSysMsgExt

fct

```
int RegisterOnSysMsgExt(HController HC,
SysMsgCallbackFunctionExt callback, void *userpointer);
```

> **NOTICE**
>
> This function is deprecated.
>
> - Use RegisterOnSystemMessage instead.

Registers a callback for a SystemMessages-event. Each time a SystemMessage occurs the callback gets called.

> **TIP**
>
> It is possible to have more than 1 callback at a time.

par **`HC`** is the handle to the controller

`callback` is the callback function

`userpointer` This pointer is passed as a parameter to the callback function

ret **`E_OK`** on success

`E_EXIST` if the callback has already been registered

`E_FAILURE` on failure

see 4.2.3.2 SysMsgCallbackFunctionExt, 4.5.12.67 UnregisterOnSysMsgSingleExt, 4.5.12.69 UnregisterOnSysMsg, 4.5.13.6 GetSysMsgRawText

4.5.12.67. UnregisterOnSysMsgSingleExt

fct

```
int UnregisterOnSysMsgSingleExt(HController HC,
SysMsgCallbackFunctionExt callback);
```

> **NOTICE**
>
> This function is deprecated.
>
> - Use UnregisterOnSystemMessageSingle instead.

Unregisters a callback for a SystemMessage-event.

par **`HC`** is the handle to the controller

`callback` is the callback function

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.12.66 RegisterOnSysMsgExt, 4.5.12.69 UnregisterOnSysMsg

4.5.12.68. RegisterOnSysMsg

fct

```
int RegisterOnSysMsg(HController HC,SysMsgCallbackFunction
callback);
```

> **NOTICE**
>
> This function is deprecated.
>
> - Use RegisterOnSystemMessage instead.

Registers a callback for a SystemMessages-event. Each time a SystemMessage occurs the callback gets called.

> **TIP**
>
> It is possible to have more than 1 callback at a time.

par **`HC`** is the handle to the controller

`callback` is the callback function

ret **`E_OK`** on success

`E_EXIST` if the callback has already been registered

`E_FAILURE` on failure

see 4.2.3.3 SysMsgCallbackFunction, 4.5.12.66 RegisterOnSysMsgExt, 4.5.12.69 UnregisterOnSysMsg, 4.5.13.6 GetSysMsgRawText

4.5.12.69. UnregisterOnSysMsg

fct

```
int UnregisterOnSysMsg(HController HC);
```

> **⚠ NOTICE**
>
> This function is deprecated.
>
> - Use UnregisterOnSystemMessage instead.

Unregisters all callbacks for a SystemMessage-event.

par **`HC`** is the handle to the controller

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.12.71 RegisterOnSystemMessage

4.5.12.70. UnregisterOnSysMsgSingle

fct

```
int UnregisterOnSysMsgSingle(HController HC,
SysMsgCallbackFunction callback);
```

Unregisters a callback for a SystemMessage-event.

NOTICE

This function is deprecated.

- Use UnregisterOnSystemMessageSingle instead.

par **`HC`** is the handle to the controller

`callback` is the callback function

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.12.68 RegisterOnSysMsg

4.5.12.71. RegisterOnSystemMessage

fct

```
int RegisterOnSystemMessage(SystemMessageCallbackFunction
callback, void *userpointer);
```

Registers a callback for a SystemMessages-event. Each time a SystemMessage occurs the callback gets called.

TIP

It is possible to have more than 1 callback at a time.

par **`callback`** is the callback function

`userpointer` This pointer is passed as a parameter to the callback function

ret `E_OK` on success

`E_EXIST` if the callback has already been registered

`E_FAILURE` on failure

see 4.2.3.1 SystemMessageCallbackFunction,
4.5.12.72 UnregisterOnSystemMessage,
4.5.12.73 UnregisterOnSystemMessageSingle,
4.5.13.2 GetSystemMessageXML

4.5.12.72. UnregisterOnSystemMessage

fct
```
int UnregisterOnSystemMessage();
```

Unregisters all callbacks for a SystemMessage-event.

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.12.71 RegisterOnSystemMessage

4.5.12.73. UnregisterOnSystemMessageSingle

fct
```
int UnregisterOnSystemMessageSingle(
SystemMessageCallbackFunction callback);
```

Unregisters a callback for a SystemMessage-event.

par `callback` is the callback function

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.12.71 RegisterOnSystemMessage,
4.5.12.72 UnregisterOnSystemMessage

4.5.12.74. RegisterOnDeviceCreated

fct

```
int RegisterOnDeviceCreated(HController HC,
DeviceCreatedCallbackFunction callback, void *userpointer);
```

Registers a callback for a DeviceCreated-event. The callback gets called when a device was created on the controller.

> **TIP**
>
> Due to design it is not ensured that the device is completely created when the callback gets called. This behavior depends on the version of the firmware.

par **`HC`** is the handle to the controller

`callback` is the callback function

`userpointer` This pointer is passed as a parameter to the callback function

ret **`E_OK`** on success

`E_EXIST` if the callback has already been registered

`E_FAILURE` on failure

see 4.5.12.75 UnregisterOnDeviceCreated,
4.5.12.76 RegisterOnDeviceDeleted

4.5.12.75. UnregisterOnDeviceCreated

fct

```
int UnregisterOnDeviceCreated(HController HC,
DeviceCreatedCallbackFunction callback);
```

Unregisters a callback for a DeviceCreated-event.

par **`HC`** is the handle to the controller

`callback` is the callback function

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.12.74 RegisterOnDeviceCreated, 4.5.12.76 RegisterOnDeviceDeleted

4.5.12.76. RegisterOnDeviceDeleted

fct
```
int RegisterOnDeviceDeleted(HController HC,
DeviceDeletedCallbackFunction callback, void *userpointer);
```

Registers a callback for a DeviceDeleted-event. The callback gets called when a device was created on the controller.

par `HC` is the handle to the controller

`callback` is the callback function

`userpointer` This pointer is passed as a parameter to the callback function

ret `E_OK` on success

`E_EXIST` if the callback has already been registered

`E_FAILURE` on failure

see 4.5.12.77 UnregisterOnDeviceDeleted,
4.5.12.74 RegisterOnDeviceCreated

4.5.12.77. UnregisterOnDeviceDeleted

fct
```
int UnregisterOnDeviceDeleted(HController HC,
DeviceDeletedCallbackFunction callback);
```

Unregisters a callback for a DeviceDeleted-event. The callback gets called when a device was created on the controller.

par `HC` is the handle to the controller

callback is the callback function

ret **E_OK** on success

E_FAILURE on failure

see 4.5.12.76 RegisterOnDeviceDeleted, 4.5.12.74 RegisterOnDeviceCreated

4.5.12.78. RegisterOnDeviceActivated

fct

```
int RegisterOnDeviceActivated(HController HC,
DeviceActivatedCallbackFunction callback, void *userpointer
);
```

par **HC** is the handle to the controller

callback is the callback function

userpointer This pointer is passed as a parameter to the callback function

ret **E_OK** on success

E_EXIST if the callback has already been registered

E_FAILURE on failure

see 4.5.12.79 UnregisterOnDeviceActivated,
4.5.12.80 RegisterOnDeviceDeactivated

4.5.12.79. UnregisterOnDeviceActivated

fct

```
int UnregisterOnDeviceActivated(HController HC,
DeviceActivatedCallbackFunction callback);
```

par **HC** is the handle to the controller

callback is the callback function

ret **E_OK** on success

E_FAILURE on failure

see 4.5.12.78 RegisterOnDeviceActivated,
4.5.12.80 RegisterOnDeviceDeactivated

4.5.12.80. RegisterOnDeviceDeactivated

fct

```
int RegisterOnDeviceDeactivated(HController HC,
DeviceDeactivatedCallbackFunction callback, void *
userpointer);
```

par **HC** is the handle to the controller

callback is the callback function

userpointer This pointer is passed as a parameter to the callback function

ret **E_OK** on success

E_EXIST if the callback has already been registered

E_FAILURE on failure

see 4.5.12.81 UnregisterOnDeviceDeactivated,
4.5.12.78 RegisterOnDeviceActivated

4.5.12.81. UnregisterOnDeviceDeactivated

fct

```
int UnregisterOnDeviceDeactivated(HController HC,
DeviceDeactivatedCallbackFunction callback);
```

par **HC** is the handle to the controller

callback is the callback function

ret **E_OK** on success

E_FAILURE on failure

see 4.5.12.78 RegisterOnDeviceActivated,
4.5.12.80 RegisterOnDeviceDeactivated

4.5.12.82. RegisterOnDeviceDependencyAdded

fct

```
int RegisterOnDeviceDependencyAdded(HController HC,
DeviceDependencyAddedCallbackFunction callback, void *
userpointer);
```

par **`HC`** is the handle to the controller

`callback` is the callback function

`userpointer` This pointer is passed as a parameter to the callback function

ret **`E_OK`** on success

`E_EXIST` if the callback has already been registered

`E_FAILURE` on failure

see 4.5.12.83 UnregisterOnDeviceDependencyAdded,
4.5.12.84 RegisterOnDeviceDependencyRemoved

4.5.12.83. UnregisterOnDeviceDependencyAdded

fct

```
int UnregisterOnDeviceDependencyAdded(HController HC,
DeviceDependencyAddedCallbackFunction callback);
```

par **`HC`** is the handle to the controller

`callback` is the callback function

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.12.84 RegisterOnDeviceDependencyRemoved,
4.5.12.82 RegisterOnDeviceDependencyAdded

4.5.12.84. RegisterOnDeviceDependencyRemoved

fct

```
int RegisterOnDeviceDependencyRemoved(HController HC,
DeviceDependencyRemovedCallbackFunction callback, void *
userpointer);
```

par **`HC`** is the handle to the controller

`callback` is the callback function

`userpointer` This pointer is passed as a parameter to the callback function

ret **`E_OK`** on success

`E_EXIST` if the callback was already registered

`E_FAILURE` on failure

see 4.5.12.85 UnregisterOnDeviceDependencyRemoved, 4.5.12.82 RegisterOnDeviceDependencyAdded

4.5.12.85. UnregisterOnDeviceDependencyRemoved

fct

```
int UnregisterOnDeviceDependencyRemoved(HController HC,
DeviceDependencyRemovedCallbackFunction callback);
```

par **`HC`** is the handle to the controller

`callback` is the callback function

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.12.82 RegisterOnDeviceDependencyAdded, 4.5.12.84 RegisterOnDeviceDependencyRemoved

4.5.12.86. RegisterOnDeviceStateChanged

fct

```
int RegisterOnDeviceStateChanged(HController HC,
DeviceStateChangedCallbackFunction callback, void *
userpointer);
```

par **`HC`** is the handle to the controller

`callback` is the callback function

`userpointer` This pointer is passed as a parameter to the callback function

ret **`E_OK`** on success

`E_EXIST` if the callback has already been registered

`E_FAILURE` on failure

see 4.5.12.87 UnregisterOnDeviceStateChanged

4.5.12.87. UnregisterOnDeviceStateChanged

fct

```
int UnregisterOnDeviceStateChanged(HController HC,
DeviceStateChangedCallbackFunction callback);
```

par **`HC`** is the handle to the controller

`callback` is the callback function

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.12.86 RegisterOnDeviceStateChanged

4.5.12.88. RegisterOnDeviceErrorStateChanged

fct

```
int RegisterOnDeviceErrorStateChanged(HController HC,
DeviceErrorStateChangedCallbackFunction callback, void *
userpointer);
```

par `HC` is the handle to the controller

`callback` is the callback function

`userpointer` This pointer is passed as a parameter to the callback function

ret `E_OK` on success

`E_EXIST` if the callback has already been registered

`E_FAILURE` on failure

see 4.5.12.89 UnregisterOnDeviceErrorStateChanged

4.5.12.89. UnregisterOnDeviceErrorStateChanged

fct

```
int UnregisterOnDeviceErrorStateChanged(HController HC,
DeviceErrorStateChangedCallbackFunction callback);
```

par `HC` is the handle to the controller

`callback` is the callback function

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.12.88 RegisterOnDeviceErrorStateChanged

4.5.12.90. RegisterOnDeviceParamStateChanged

fct

```
int RegisterOnDeviceParamStateChanged(HController HC,
DeviceParamStateChangedCallbackFunction callback, void *
userpointer);
```

par **`HC`** is the handle to the controller

`callback` is the callback function

`userpointer` This pointer is passed as a parameter to the callback function

ret **`E_OK`** on success

`E_EXIST` if the callback has already been registered

`E_FAILURE` on failure

see 4.5.12.91 UnregisterOnDeviceParamStateChanged

4.5.12.91. UnregisterOnDeviceParamStateChanged

fct

```
int UnregisterOnDeviceParamStateChanged(HController HC,
DeviceParamStateChangedCallbackFunction callback);
```

par **`HC`** is the handle to the controller

`callback` is the callback function

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.12.90 RegisterOnDeviceParamStateChanged

4.5.12.92. RegisterOnDevicePowerStateChanged

fct

```
int RegisterOnDevicePowerStateChanged(HController HC,
DevicePowerStateChangedCallbackFunction callback, void *
userpointer);
```

par **`HC`** is the handle to the controller

`callback` is the callback function

`userpointer` This pointer is passed as a parameter to the callback function

ret **`E_OK`** on success

`E_EXIST` if the callback has already been registered

`E_FAILURE` on failure

see 4.5.12.93 UnregisterOnDevicePowerStateChanged

4.5.12.93. UnregisterOnDevicePowerStateChanged

fct

```
int UnregisterOnDevicePowerStateChanged(HController HC,
DevicePowerStateChangedCallbackFunction callback);
```

par **`HC`** is the handle to the controller

`callback` is the callback function

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.12.92 RegisterOnDevicePowerStateChanged

4.5.12.94. RegisterOnJobLines

fct

```
int RegisterOnJobLines(HController HC,
JobLinesCallbackFunction callback);
```

par **`HC`** is the handle to the controller

`callback` is the callback function

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.12.95 UnregisterOnJobLinesSingle, 4.5.12.96 UnregisterOnJobLines

4.5.12.95. UnregisterOnJobLinesSingle

fct

```
int UnregisterOnJobLinesSingle(HController HC,
JobLinesCallbackFunction callback);
```

par **`HC`** is the handle to the controller

`callback` is the callback function

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.12.94 RegisterOnJobLines, 4.5.12.96 UnregisterOnJobLines

4.5.12.96. UnregisterOnJobLines

fct

```
int UnregisterOnJobLines(HController HC);
```

par **`HC`** is the handle to the controller

`callback` is the callback function

ret **`E_OK`** on success

E_FAILURE on failure

see 4.5.12.94 RegisterOnJobLines, 4.5.12.95 UnregisterOnJobLinesSingle

4.5.12.97. RegisterOnTssData

fct

```
int RegisterOnTssData(HController HC,
TssDataCallbackFunction callback, void *userpointer);
```

par **HC** is the handle to the controller

callback is the callback function

userpointer This pointer will be a parameter in every callback of this type

ret **E_OK** on success

E_FAILURE on failure

see 4.5.12.98 UnregisterOnTssDataSingle, 4.5.12.99 UnregisterOnTssData

4.5.12.98. UnregisterOnTssDataSingle

fct

```
int UnregisterOnTssDataSingle(HController HC,
TssDataCallbackFunction callback);
```

par **HC** is the handle to the controller

callback is the callback function

ret **E_OK** on success

E_FAILURE on failure

see 4.5.12.97 RegisterOnTssData, 4.5.12.99 UnregisterOnTssData

4.5.12.99. UnregisterOnTssData

fct

```
int UnregisterOnTssData(HController HC);
```

par **`HC`** is the handle to the controller

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.12.97 RegisterOnTssData, 4.5.12.98 UnregisterOnTssDataSingle

4.5.12.100. RegisterOnUpdateTssChannels

fct

```
int RegisterOnUpdateTssChannels(HController HC,
UpdateTssChannelsCallbackFunction callback, void *
userpointer);
```

par **`HC`** is the handle to the controller

`callback` is the callback function

`userpointer` This pointer will be a parameter in every callback of this type

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.12.101 UnregisterOnUpdateTssChannelsSingle,
4.5.12.102 UnregisterOnUpdateTssChannels

4.5.12.101. UnregisterOnUpdateTssChannelsSingle

fct

```
int UnregisterOnUpdateTssChannelsSingle(HController HC,
UpdateTssChannelsCallbackFunction callback);
```

par **`HC`** is the handle to the controller

`callback` is the callback function

ret **E_OK** on success

E_FAILURE on failure

see 4.5.12.100 RegisterOnUpdateTssChannels,
4.5.12.102 UnregisterOnUpdateTssChannels

4.5.12.102. UnregisterOnUpdateTssChannels

fct

```
int UnregisterOnUpdateTssChannels(HController HC);
```

par **HC** is the handle to the controller

ret **E_OK** on success

E_FAILURE on failure

see 4.5.12.100 RegisterOnUpdateTssChannels,
4.5.12.101 UnregisterOnUpdateTssChannelsSingle

4.5.13. System messages

Table of contents

4.5.13.1. GetSystemMessageXMLLen 264
4.5.13.2. GetSystemMessageXML 265
4.5.13.3. GetSysMsgXMLLen 268
4.5.13.4. GetSysMsgXML 268
4.5.13.5. GetSysMsgRawTextLen 272
4.5.13.6. GetSysMsgRawText 273

4.5.13.1. GetSystemMessageXMLLen

fct

```
int GetSystemMessageXMLLen(HSysMsg HSM);
```

Gets the length of a given SysMsg as XML.

> **TIP**
>
> This call is only valid from inside a SysMsgCallbackFunction.

par `HSM` is the handle to the SysMsg

ret the strlen of the SysMsg without the trailing 0-byte

`E_FAILURE` on failure

see 4.5.13.2 GetSystemMessageXML, 4.2.3.3 SysMsgCallbackFunction

4.5.13.2. GetSystemMessageXML

fct

```
int GetSystemMessageXML(HSysMsg HSM, char *buffer, int
bufferlen);
```

Gets the XML-text of a SysMsg. Call GetSysMsgXMLLen to get the buffer-length. If the function returns `E_OK` the `buffer` holds the `NULL`-terminated string. To work correctly the buffer length has to be `GetSysMsgXMLLen+1`. SysMessages are divided into contexts and messages. Contexts are optional.

Listing 4.41: SysMessage with neither context nor parameters

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<SYSMESSAGE ID="3" ControllerHandle="1" Date="2010-11-17
  " Time="07:51:27" Uptime="122.232921">
  <MESSAGE Name="FW_E_BREAK" Caller="HandleSysMsg"
     Facility="Firmware"/>
</SYSMESSAGE>
```

Listing 4.42: SysMessage with 1 context and 1 message with parameters

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<SYSMESSAGE ID="1" ControllerHandle="1" Date="2010-11-17
  " Time="07:51:27" Uptime="122.232453">
  <CONTEXT Name="Execute Script usr.job.Job.Script">
    <MESSAGE Name="FW_SCRIPT_ERROR" Caller="HandleSysMsg
      " Facility="Firmware">
      <PARAM Name="ERRORDESCRIPTION" Value="syntax error
        "/>
      <PARAM Name="LINE" Value="1"/>
      <PARAM Name="COLUMN" Value="9"/>
    </MESSAGE>
  </CONTEXT>
</SYSMESSAGE>
```

Listing 4.43: SysMessage with 3 contexts and 1 message

```xml
<?xml version="1.0" encoding="UTF-8" ?>
```

```
<SYSMESSAGE ID="1" ControllerHandle="1" Date="2010-11-17
   " Time="07:51:27" Uptime="122.232453">
  <CONTEXT Name="Execute Conditional usr.job.Job.
     Conditional">
    <CONTEXT Name="Execute Conditional usr.job.Job.
       Conditional.Conditional">
      <CONTEXT Name="Execute Conditional usr.job.Job.
         Conditional.Conditional.Script">
        <MESSAGE Name="FW_SCRIPT_ERROR" Caller="
           HandleSysMsg" Facility="Firmware">
          <PARAM Name="ERRORDESCRIPTION" Value="syntax
             error"/>
          <PARAM Name="LINE" Value="1"/>
          <PARAM Name="COLUMN" Value="9"/>
        </MESSAGE>
      </CONTEXT>
    </CONTEXT>
  </CONTEXT>
</SYSMESSAGE>
```

The `SYSMESSAGE` tag can hold `CONTEXT` or `MESSAGE` child tags. The attribute `ID` is a unique identifier for the complete message. The attribute `ControllerHandle` gives the handle to the controller if the message can be dedicated to a controller, otherwise it is omitted. The attribute `Date` gives the date the SysMessage occured in the form `yyyy-mm-dd`. The attribute `Time` gives the time the SysMessage occured in the form `hh:mm:ss`. The attribute `Uptime` gives the time the ControllerLib application is running in seconds.

A `CONTEXT` tag can hold another `CONTEXT` or a `MESSAGE` child tag. Its only attribute `Name` gives the name of the context.

A `MESSAGE` tag can hold a variable count of `PARAM` tags. The attribute `Name` gives the name of the SysMessage. This is the primary key in a database where the complete text of the message can be found. The attribute `Caller` tells, where the SysMessage was thrown. Normally, this is the function name. The attribute `Facility` tells, in which facility the SysMessage was thrown. This can be `Firmware`, `ControllerLib` or `ClientApplication`.

A `PARAM` tag can hold any child tags. It is used to fill out variables in the complete message text found in the database. The attribute `Name` gives the name of the variable to change and the attribute `Value` gives the value the variable should be changed to.

Listing 4.44: GetSystemMessageXML example

```
int buflen = GetSysMsgXMLLen(HC, HSM);
if (buflen > 0) {
  char *buffer = (char*)malloc(++buflen);
  if (GetSysMsgXML(HC, HSM, buffer, buflen) == E_OK) {
    printf("MessageXML:\n %s\n", buffer);
  }
  free(buffer);
}
```

TIP

This call is only valid from inside a SysMsgCallbackFunction.

par **`HSM`** is the handle to the SysMsg

`buffer` is the pointer to the buffer for the SysMsg

`bufferlen` is the size of the buffer in bytes

ret **`E_OK`** on success (*buffer* holds the string)

`E_NOSPACE` if the buffer was too small

`E_UNAVAIL` if the message is not available

`E_FAILURE` on failure

see 4.5.13.1 GetSystemMessageXMLLen, 4.2.3.3 SysMsgCallbackFunction

4.5.13.3. GetSysMsgXMLLen

fct

```
int GetSysMsgXMLLen(HController HC, HSysMsg HSM);
```

NOTICE

This function is deprecated.

- Use GetSystemMessageXMLLen instead.

Gets the length of a given SysMsg as XML.

TIP

This call is only valid from inside a SysMsgCallbackFunction.

par **`HC`** is the handle to the controller

`HSM` is the handle to the SysMsg

ret the strlen of the SysMsg without the trailing 0-byte

`E_FAILURE` on failure

see 4.5.13.5 GetSysMsgRawTextLen, 4.2.3.3 SysMsgCallbackFunction

4.5.13.4. GetSysMsgXML

fct

```
int GetSysMsgXML(HController HC, HSysMsg HSM, char *buffer,
 int bufferlen);
```

NOTICE

This function is deprecated.

- Use GetSystemMessageXML instead.

Gets the XML text of a SysMsg. Call GetSysMsgXMLLen to get the buffer length. If the function returns `E_OK` the `buffer` holds the `NULL`-terminated string. To work correctly the buffer length has to be `GetSysMsgXMLLen+1`.

SysMessages are divided in contexts and messages. Contexts are optional.

Listing 4.45: SysMessage with neither context nor parameters

```
<?xml version="1.0" encoding="UTF-8" ?>
<SYSMESSAGE ID="3" Date="2010-11-17" Time="07:51:27"
   Uptime="122.232921">
  <MESSAGE Name="FW_E_BREAK" Caller="HandleSysMsg"
     Facility="Firmware"/>
</SYSMESSAGE>
```

Listing 4.46: SysMessage with 1 context and 1 message with parameters

```
<?xml version="1.0" encoding="UTF-8" ?>
<SYSMESSAGE ID="1" Date="2010-11-17" Time="07:51:27"
   Uptime="122.232453">
  <CONTEXT Name="Execute Script usr.job.Job.Script">
    <MESSAGE Name="FW_SCRIPT_ERROR" Caller="HandleSysMsg
       " Facility="Firmware">
      <PARAM Name="ERRORDESCRIPTION" Value="syntax error
         "/>
      <PARAM Name="LINE" Value="1"/>
      <PARAM Name="COLUMN" Value="9"/>
    </MESSAGE>
  </CONTEXT>
</SYSMESSAGE>
```

Listing 4.47: SysMessage with 3 contexts and 1 message

```
<?xml version="1.0" encoding="UTF-8" ?>
<SYSMESSAGE ID="1" Date="2010-11-17" Time="07:51:27"
   Uptime="122.232453">
  <CONTEXT Name="Execute Conditional usr.job.Job.
     Conditional">
    <CONTEXT Name="Execute Conditional usr.job.Job.
       Conditional.Conditional">
      <CONTEXT Name="Execute Conditional usr.job.Job.
         Conditional.Conditional.Script">
```

```
          <MESSAGE Name="FW_SCRIPT_ERROR" Caller="
             HandleSysMsg" Facility="Firmware">
            <PARAM Name="ERRORDESCRIPTION" Value="syntax
               error"/>
            <PARAM Name="LINE" Value="1"/>
            <PARAM Name="COLUMN" Value="9"/>
          </MESSAGE>
        </CONTEXT>
      </CONTEXT>
    </CONTEXT>
  </SYSMESSAGE>
```

The `SYSMESSAGE` tag can hold `CONTEXT` or `MESSAGE` child tags. The attribute `ID` is a unique identifier for the complete message. The attribute `Date` gives the date the SysMessage occured in the form `yyyy-mm-dd`. The attribute `Time` gives the time the SysMessage occured in the form `hh:mm:ss`. The attribute `Uptime` gives the time the ControllerLib application is running in seconds.

A `CONTEXT` tag can hold another `CONTEXT` or a `MESSAGE` child tag. Its only attribute `Name` gives the name of the context.

A `MESSAGE` tag can hold a variable count of `PARAM` tags. The attribute `Name` gives the name of the SysMessage. This is the primary key in a database where the complete text of the message can be found. The attribute `Caller` tells, where the SysMessage was thrown. Normally, this is the function name. The attribute `Facility` tells, in which facility the SysMessage was thrown. This can be `Firmware`, `ControllerLib` or `ClientApplication`.

A `PARAM` tag can hold any child tags. It is used to fill out variables in the complete message text found in the database. The attribute `Name` gives the name of the variable to change and the attribute `Value` gives the value the variable should be changed to.

Listing 4.48: GetSysMsgXML example

```
int buflen = GetSysMsgXMLLen(HC, HSM);
if (buflen > 0) {
```

```
    char *buffer = (char*)malloc(++buflen);
    if (GetSysMsgXML(HC, HSM, buffer, buflen) == E_OK) {
      printf("MessageXML:\n %s\n", buffer);
    }
    free(buffer);
  }
```

TIP

This call is only valid from inside a SysMsgCallbackFunction.

par **HC** is the handle to the controller

HSM is handle to the SysMsg

buffer is the pointer to the buffer for the SysMsg

bufferlen is the size of the buffer in bytes

ret **E_OK** on success (*buffer* holds the string)

E_NOSPACE if the buffer was too small

E_UNAVAIL if the message is not available

E_FAILURE on failure

see 4.5.13.3 GetSysMsgXMLLen, 4.5.13.6 GetSysMsgRawText, 4.2.3.3 SysMsgCallbackFunction

4.5.13.5. GetSysMsgRawTextLen

fct

```
int GetSysMsgRawTextLen(HController HC, HSysMsg HSM);
```

⚠ NOTICE

This function is deprecated.

- Use GetSystemMessageXMLLen instead.

Gets the length of a given SysMsg.

> **TIP**
>
> This call is only valid from inside a SysMsgCallbackFunction.

par **`HC`** is the handle to the controller

`HSM` is the handle to the SysMsg

ret the strlen of the SysMsg without the trailing 0-byte

`E_FAILURE` on failure

see 4.5.13.6 GetSysMsgRawText, 4.5.13.3 GetSysMsgXMLLen, 4.5.13.4 GetSysMsgXML, 4.2.3.3 SysMsgCallbackFunction

4.5.13.6. GetSysMsgRawText

fct

```
int GetSysMsgRawText(HController HC, HSysMsg HSM, char *
buffer, int bufferlen);
```

Gets the raw text of a SysMsg. Call GetSysMsgRawTextLen to get the buffer length. If the function returns `E_OK` the `buffer` holds the `NULL`-terminated string. To work correctly the buffer length has to be `GetSysMsgRawTextLen+1`. Messages are divided in contexts and messages. Each context and each message have a `NAME` and `LEVEL`. The `NAME` identifies the context or message and the `LEVEL` gives the hierarchical information. Message blocks also hold information about the number of parameters and the parameters itself. Context blocks are optional. Each block can occur multiple times.

Listing 4.49: Context blocks are of this form

```
CONTEXTBEGIN
  NAME=<string>
  LEVEL=<number>
CONTEXTEND
```

Listing 4.50: Message blocks are of this form

```
MESSAGEBEGIN
  NAME=<string>
  LEVEL=<number>
  ARGC=<number>
  <string>=<variant>
MESSAGEEND
```

Listing 4.51: Complete message string example

```
CONTEXTBEGIN
  NAME=Execute Script usr.job.Job.Script
  LEVEL=0
CONTEXTEND
MESSAGEBEGIN
  NAME=FW_SCRIPT_ERROR
  LEVEL=1
  ARGC=3
  ERRORDESCRIPTION=missing 'C'
  LINE=1
  COLUMN=3
MESSAGEEND
MESSAGEBEGIN
  NAME=FW_SCRIPT_ERROR
  LEVEL=1
  ARGC=3
  ERRORDESCRIPTION=syntax error
  LINE=1
  COLUMN=3
MESSAGEEND
```

Listing 4.52: GetSysMsgRawText call example

```
int buflen = GetSysMsgRawTextLen(HC, HSM);
if (buflen > 0) {
  char *buffer = (char*)malloc(++buflen);
  if (GetSysMsgRawText(HC, HSM, buffer, buflen) == E_OK)
      {
```

```
        printf("Messagerawtext: %s\n", buffer);
    }
    free(buffer);
}
```

> **TIP**
>
> This call is only valid from inside a SysMsgCallbackFunction.

par **`HC`** is the handle to the controller

`HSM` is the handle to the SysMsg

`buffer` is the pointer to the buffer for the SysMsg

`bufferlen` is the size of the buffer in bytes

ret **`E_OK`** on success (`buffer` holds the string)

`E_NOSPACE` if the buffer was too small

`E_UNAVAIL` if the message is not available

`E_FAILURE` on failure

see 4.5.13.5 GetSysMsgRawTextLen, 4.5.13.4 GetSysMsgXML,
4.2.3.3 SysMsgCallbackFunction

4.5.14. (Scan) heads

Table of contents

4.5.14.1. GetHeadCount 276
4.5.14.2. GetHead 276

4.5.14.1. GetHeadCount

fct

```
int GetHeadCount(HController HC);
```

Gets the number of (scan) heads managed by the controller.

par **`HC`** is the handle to the controller

ret the number of (scan) heads

`E_FAILURE` on failure

see 4.5.14.2 GetHead

4.5.14.2. GetHead

fct

```
HHead GetHead(HController HC, int index);
```

Gets a specific (scan) head. Information about (scan) heads is needed for distortion or scan field correction.

par **`HC`** is the handle to the controller

`index` is the index of the (scan) head. The index starts at `0`

ret the handle to the (scan) head

`ARG_INVALID_HANDLE_VALUE` on failure

see 4.5.14.1 GetHeadCount

4.5.15. Distortion and scan field correction

Table of contents

4.5.15.1. LoadDistortion 277
4.5.15.2. SaveDistortion 277
4.5.15.3. GetActiveScanfieldCorrection 278
4.5.15.4. ImproveScanfieldCorrection 278

4.5.15.1. LoadDistortion

fct

```
int LoadDistortion(HController HC, HHead HH, const char *
filename);
```

Loads a distortion file to the controller. Distortion files normally have the extension `dst`.

par `HC` is the handle to the controller

`HH` is the handle to the (scan) head

`filename` is the file with distortion data

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.15.2 SaveDistortion

4.5.15.2. SaveDistortion

fct

```
int SaveDistortion(HController HC, HHead HH, const char *
filename);
```

Saves the current distortion to a file. Distortion files normally have the extension `dst`.

par `HC` is the handle to the controller

HH is the handle to the (scan) head

filename is the file for saving the distortion data

ret **E_OK** on success

E_FAILURE on failure

see 4.5.15.1 LoadDistortion

4.5.15.3. GetActiveScanfieldCorrection

fct

```
HScanfieldCorrection GetActiveScanfieldCorrection(
HController HC, HHead HH);
```

Gets the active scan field correction for the given head.

par **HC** is the handle to the controller

HH is the handle to the (scan) head

ret the handle to the active scan field correction

ARG_INVALID_HANDLE_VALUE on failure

4.5.15.4. ImproveScanfieldCorrection

fct

```
int ImproveScanfieldCorrection(HController HC,
HScanfieldCorrection HSC, char *AEDfilename);
```

Improves a given scan field correction.

par **HC** is the handle to the controller

HSC is the handle to the scan field correction

AEDfilename is the name of the AED-file

ret **E_OK** on success

E_FAILURE on failure

4.5.16. Logging

A request log records the communication between the ARGES system controller and the ControllerLib. This is useful for debugging the protocol and only specialized applications should make use of this.

Table of contents

4.5.16.1. RegisterOnRequestLog 279
4.5.16.2. UnregisterOnRequestLog 280
4.5.16.3. SetRequestLogFilename 280
4.5.16.4. SetLogLevel 281
4.5.16.5. GetLogLevel 281
4.5.16.6. RegisterOnLog 282
4.5.16.7. UnregisterOnLog 282
4.5.16.8. SetLogFilename 283
4.5.16.9. Log 283
4.5.16.10. StartTssRecorder 284
4.5.16.11. StopTssRecorder 284

4.5.16.1. RegisterOnRequestLog

fct

```
int RegisterOnRequestLog(RequestLogCallbackFunction
callback, void *userpointer);
```

Registers a callback for RequestLog-events. Only 1 callback of this type is allowed. To write the request log to a file call SetRequestLogFilename. To record all requests from the beginning of the communication between controller and library call this function before calling DetectRemoteController.

par **`callback`** is the callback function

`userpointer` is a parameter in the callback function

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.16.3 SetRequestLogFilename, 4.5.2.2 DetectRemoteController, 4.5.16.2 UnregisterOnRequestLog, 4.2.5.2 RequestLogCallbackFunction

4.5.16.2. UnregisterOnRequestLog

fct

```
int UnregisterOnRequestLog();
```

Unregisters a callback for RequestLog-events.

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.16.1 RegisterOnRequestLog, 4.5.16.3 SetRequestLogFilename, 4.2.5.2 RequestLogCallbackFunction

4.5.16.3. SetRequestLogFilename

fct

```
int SetRequestLogFilename(const char *filename, int
deletefirst, int maxSizeInMB);
```

Sets the name of the file where the RequestLog should be written to. Please note that it is possible to have the request log written to a file and also have a callback for this.

par **`filename`** is the complete file name for the request log, e.g. `/tmp/request.log`. To stop logging pass `NULL` as file name

`deletefirst` deletes the file during this call if it is set to `1`

`maxSizeInMB` limits the file size. If this is set to `0` the limit is removed

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.16.2 UnregisterOnRequestLog, 4.2.5.2 RequestLogCallbackFunction

4.5.16.4. SetLogLevel

fct
```
int SetLogLevel(ARG_LOG_LEVEL level);
```

Sets the logging level. All logging-events which are less than the current log level are supressed. Valid log levels are:
LOG_SUCCESS < LOG_INFO < LOG_WARNING < LOG_ERROR

par `level` is the new log level

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.16.5 GetLogLevel, 4.5.16.8 SetLogFilename, 4.5.16.6 RegisterOnLog, 4.5.16.9 Log

4.5.16.5. GetLogLevel

fct
```
ARG_LOG_LEVEL GetLogLevel();
```

Gets the logging level. All logging-events which are less than the current log level are supressed. Valid log levels are:
LOG_SUCCESS < LOG_INFO < LOG_WARNING < LOG_ERROR

ret current log level

see 4.5.16.5 GetLogLevel, 4.5.16.8 SetLogFilename, 4.5.16.6 RegisterOnLog, 4.5.16.9 Log

4.5.16.6. RegisterOnLog

fct

```
int RegisterOnLog(LogCallbackFunction callback, void *
userpointer);
```

Registers a callback for log-events. The client application gets internal log messages of the ControllerLib as well as messages written with Log, if the log level is not filtered. (SetRequestLogFilename).

par **`callback`** is the callback function

`userpointer` is a parameter in the callback function

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.16.7 UnregisterOnLog, 4.5.16.8 SetLogFilename, 4.5.16.9 Log, 4.5.16.4 SetLogLevel, 4.5.16.5 GetLogLevel, 4.2.5.3 LogCallbackFunction

4.5.16.7. UnregisterOnLog

fct

```
int UnregisterOnLog();
```

Unregisters a callback for log-events.

par **`callback`** is the callback function

`userpointer` is a parameter in the callback function

ret **`E_OK`** on success

`E_FAILURE` on failure

see 4.5.16.6 RegisterOnLog, 4.5.16.8 SetLogFilename, 4.5.16.9 Log, 4.5.16.4 SetLogLevel, 4.5.16.5 GetLogLevel, 4.2.5.3 LogCallbackFunction

4.5.16.8. SetLogFilename

fct
```
int SetLogFilename(const char *filename);
```

Sets the name of the file where a log file should be written to. Please note that it is possible to have a log written to a file and have also a callback for it.

par `callback` is the callback function

`userpointer` is a parameter in the callback function

`filename` is the complete file name for the request log, e.g. `/tmp/request.log`. To stop logging pass `NULL` as file name

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.16.7 UnregisterOnLog, 4.5.16.8 SetLogFilename, 4.2.5.3 LogCallbackFunction, 4.5.16.4 SetLogLevel, 4.5.16.5 GetLogLevel, 4.5.16.9 Log

4.5.16.9. Log

fct
```
int Log(ARG_LOG_LEVEL level, const char *fmt, ...);
```

Use this function to log the flow in the client application. The function uses the internal logging of the ControllerLib and makes use of the callback or the log file. It can be used like `printf`.

par `level` is the log level

`fmt` is the format string

`...` are optional parameters

ret `E_OK` on success

`E_FAILURE` on failure

see 4.5.16.7 UnregisterOnLog, 4.5.16.8 SetLogFilename, 4.2.5.3 LogCallbackFunction, 4.5.16.4 SetLogLevel, 4.5.16.5 GetLogLevel

4.5.16.10. StartTssRecorder

fct

```
int StartTssRecorder(HController HC, char **channelnames,
int channelcount, ARG_TSSREC_TYPE _type, const char* path);
```

par

ret `E_OK` on success

`E_FAILURE` on failure

4.5.16.11. StopTssRecorder

fct

```
int StopTssRecorder();
```

par

ret `E_OK` on success

`E_FAILURE` on failure

5. Semantics of the interface

An ARGES system controller works based on jobs. The term *job* is defined as a process or program, that can be designed by using the ARGES InScript software. A job is a sequence of commands that controls devices that are connected to the controller, evaluates external inputs, etc.

A jobs consists of *nodes*, which are hierarchically related to each other and are displayed as a tree structures in the InScript software. Nodes containing other nodes are called parent nodes of the latter ones which are called child nodes in turn. There are different types of nodes. Each type has a function and properties. Child nodes may inherit properties of their parent nodes. The process or program, and thus the wanted output, results from the arrangement of the nodes within the tree structure. The ARGES ControllerLib handles all aspects of these items.

On special cases the controller passes SysMessages to the user. The controller sends these messages to inform or to warn about events on the controller or to give additional information on errors.

TIP

The examples in this chapter *do not* check whether the function calls are successful or not. They concentrate on the current problem. So it normally is assumed that the connection to the controller is already established.

5.1. Working with jobs

The following subsections show how to get a list of jobs present on the ARGES system controller and how to load a job.

5.1.1. Getting a list of jobs

To find out, which jobs are present on the ARGES system controller, 4.5.9.4 GetJobNames returns a list of JobNames seperated by semicolons. To get notified, when a new job is created or loaded on the controller, register on a NodeCreated event with 4.5.12.27 RegisterOnNodeCreated and check for new nodes under `usr.job`. Another convenient way of obtaining information about jobs is 4.5.9.1 GetJobInfo. This call returns a 4.1.2.3 ARG_JOBINFO-structure which has the number of jobs on the controller, the names and also the selected node.

5.1.2. Loading a job

To load a job from the file system to the controller the function 4.5.6.1 LoadJob can be called. It is possible with this call to clear other jobs on the controller first or select the loaded one immediately for execution. Also job dependencies will be loaded as pens, fonts, etc. This is the search strategy for pens. The same search strategy applies to fonts:

- With PenPath
  - LoadJob
    - File system on the controller
    - PenPath
  - Upload-Before-Execution
    - File system on the controller
    - PenPath
- Without PenPath

- LoadJob
  * File system on the controller
  * The directory, where the job is loaded from
- Upload-Before-Execution
  * File system on the controller
  * The current directory of the application

TIP

When working with Pens or Fonts, always use 4.5.5.1 SetPenPath and 4.5.5.3 SetFontPath to define where a pen is loaded from.

5.2. Working with NodeObjects

A NodeObject represents a single node from the variable tree on the ARGES system controller. In the ARGES InScript software this variable structure can be viewed with the *Inspector*. To copy the complete name to the clipboard right-click the variable and click **Fullpath to clipboard**. Then the name can be pasted into the source code. Other information about the node can also be found in the Inspector, as e.g. types, min-values, max-values.

The user can create user variables in subtree `usr.var`. Though it is possible to create variables under `usr.job`, it is not recommended as this is the structure where jobs are loaded to. But it is usual to read nodes from subtree `usr.job`.

There are 2 ways to work with variables:

- Get each node seperatly from the controller

– Use a VariableCache which mirrors all variables from the controller

TIP

To gain maximum performance it is recommended to use the VariableCache, which can be activated with a call to 4.5.2.6 EnableVariableCache right after 4.5.2.2 DetectRemoteController. Although this increases the start-up time the improved performance during run-time prevails.

5.2.1. Reading variable values

To read values of variable a handle to a NodeObject has to be obtained. In this example we get the name of the ARGES system controller, which is stored as `VAR:STRING` in variable `sys.BoardName`.

Listing 5.1: Reading a variable value example

```
HNodeObject HNO = GetNode(HC,"sys.BoardName");
ReadNode(HC, HNO); // If VariableCache is enabled this line
    can be ommited
int buflen = GetNodeValueStringLen(HC, HNO);
char *boardname = (char*)malloc(++buflen);
GetNodeValueString(HC, HNO, boardname, buflen);
```

After this sequence `boardname` holds the name of the controller.
4.5.7.82 GetNodeValueString works for any type of variable on the controller.
But there are also dedicated getter methods for variable types, like e.g.
4.5.7.80 GetNodeValueBool, 4.5.7.83 GetNodeValueInt32,
4.5.7.82 GetNodeValueString.

For a complete example that demonstrates connecting to a controller and error checks see 5.5.2 Reading a variable.

5.2.2. Writing variable values

In order to change the value of a variable, it has to be loaded by the ControllerLib from the ARGES system controller first.

Listing 5.2: Setting a new value for NodeObject sys.BoardName example

```
HNodeObject HNO = GetNode(HC,"sys.BoardName");
ReadNode(HC, HNO); // If VariableCache is enabled this line
    can be ommited
SetNodeValueString(HC, HNO, "New Board Name");
WriteNode(HC, HNO);
```

After this sequence the name of the controller will change to "`New Board Name`". 4.5.7.87 SetNodeValueString works for most variable types on the controller. But there are also dedicated setter methods for variable types, like e.g. 4.5.7.79 SetNodeValueBool, 4.5.7.88 SetNodeValueInt32, 4.5.7.87 SetNodeValueString.

For a more detailed example that demonstrates connecting to a controller and error checks see 5.5.3 Writing a variable.

5.2.3. Working with value change callbacks

Instead of polling variables, which might change in value, the ControllerLib also provides a mechanism that calls a function, when a variable changes its value. This can be done for single NodeObjects with 4.5.12.10 RegisterOnValueChanged. For a larger number of NodeObjects the 4.5.8 NodeObjectCollections should be used to increase the performance. NodesObjects which have ValueChangeCallbacks attached to them do not have to call 4.5.7.6 ReadNode before setting a new value.

5.2.4. Creating user variables

User variables can be created with calls to 4.5.7.30 CreateNodeOnController. The ControllerLib also allows creation of nodes in each subtree on the ARGES system controller. User variables should reside under `usr.var`, e.g.:

```
CreateNodeOnController(HC, "usr.var.test", VT_BOOL, 0);
```

This call creates a node on the controller. If a callback for node creation was registered with 4.5.12.27 RegisterOnNodeCreated this function gets called. If the VariableCache was enabled with 4.5.2.6 EnableVariableCache then the new node is also available in the cache.

5.2.5. Getting information about NodeObjects

NodeObjects can also have a minimum and a maximum value. These values are set by the ARGES system controller firmware and are read-only. The same applies to optional units.

5.3. Working with NodeObjectCollections

NodeObjectCollections hold references to a subset of NodeObjects. Reading or writing complete NodeObjectCollections usually is a lot faster for larger numbers of NodeObjects. Also NodeObjectCollections are used if the VariableCache is enabled with 4.5.2.6 EnableVariableCache to get the subnodes of a node.

TIP

When using more than 10 variables it is advisable to use NodeObjectCollections or to enable the VariableCache with 4.5.2.6 EnableVariableCache.

The following subsections show how to create and destroy (delete) NodeObjectCollections and how to use them.

5.3.1. Creating and destroying NodeObjectCollections

NodeObjectCollections can be created by calling
4.5.8.1 CreateNodeObjectCollection and deleted by calling

4.5.8.2 DestroyNodeObjectCollection. When destroying a NodeObjectCollection all contained NodeObjects get also destroyed.

5.3.2. Using NodeObjectCollections

To derive the full advantage from the better performance when using NodeObjectCollections, it is recommended to use
4.5.8.4 AddNodeObjectByName for adding NodeObjects to the collection.

Callback functions will not be automatically attached. This has to be done manually after a call to 4.5.8.10 ReadNodeObjectCollection.

Listing 5.3: Using NodeObjectCollections example

```
HNodeObjectCollection HNOC = CreateNodeObjectCollection(HC);
AddNodeObjectByName(HC,HNOC,"usr.var.x");
AddNodeObjectByName(HC,HNOC,"usr.var.y");
ReadNodeObjectCollection(HC, HNOC);
HNodeObject HNO_usr_var_x = GetNode(HC, "usr.var.x");
HNodeObject HNO_usr_var_y = GetNode(HC, "usr.var.y");
SetNodeValueInt32(HC, HNO_usr_var_x, 10);
SetNodeValueInt32(HC, HNO_usr_var_y, 20);
WriteNodeObjectCollection(HC, HNOC);
DestroyNodeObjectCollection(HC, HNOC);
DestroyNodeObject(HNO_usr_var_x);
DestroyNodeObject(HNO_usr_var_y);
```

5.4. Using the VariableCache

The VariableCache mirrors all nodes on the ARGES system controller to the ControllerLib. When new nodes are created on the controller they are also mapped automatically and when nodes get deleted, removed or renamed this is also mirrored. It is also possible to walk through the tree structure of the nodes. This tree structure can be viewed in the ARGES InScript software with the *Inspector*. The advantage of this approach is that the current value of the nodes on the controller always are in the variables. Although the call to 4.5.2.6 EnableVariableCache may take a little time – up to a few seconds depending on the network and on the hardware in use – this is much faster than reading every single node seperatly. Also the NodeObjects are destroyed automatically when 4.5.2.10 DisconnectController is called.

Listing 5.4: Using the VariableCache example

```
HController HC = DetectRemoteController("192.168.1.42",1610);
EnableVariableCache(HC); // This call might take a little

// Getting the parent node of usr.var.x
HNodeObject HNO_usr_var_x = GetNode(HC, "usr.var.x");
HNodeObject HNO_parent = GetParentNode(HC, HNO_usr_var_x);
// HNO_parent is now "usr.var"

// And now print all subnodes of usr.var
HNodeObjectCollection HNOC = CreateNodeObjectCollection(HC);
GetSubnodes(HC, HNO_parent, HNOC);
int cnt;
GetNodeObjectCount(HC, HNOC, &cnt);
for (int i=0; i<cnt; ++i) {
   HNodeObject HNO;
   GetNodeObjectAtIndex(HC, HNOC, i, &HNO);
   int len = GetNodeNameLen(HC, HNO);
   char *str = (char*)malloc(++len);
   GetNodeName(HC, HNO, str, len);
   printf("%s", str);
   free(str);
}
DestroyNodeObjectCollection(HC, HNOC);
```

5.5. Source code examples

The following subsections show how to link and compile a code example and list code examples for reading, writing and watching a variable as well as for loading and marking a job.

5.5.1. Linking and compiling an example

Requirements

- ☐ GNU c++ or similar compiler
- ☐ The ControllerLib has to be available in the `LD_LIBRARY_PATH`

Procedure

- Link and compile, e.g. `ReadVariable.c`:

```
g++ -o ReadVariable -lcontrollerlib -lpthread
    ReadVariable.c
```

5.5.2. Reading a variable

Listing 5.5: Reading a variable example

```
/**
 * Connect to a controller and print the value of a variable.
 *
 * For example:
 *    ./readVariable 192.168.1.42 stat.time.TimeStr
 */

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include ”arg_controllerlib.h”

#define DEFAULT_PORT 1610

/**
 * Print out usage information
 */
void usage(const char *progname)
{
```

```
    printf("\n");
    printf("Prints the value of a variable\n");
    printf("\n");
    printf("Usage:\n");
    printf("  %s <IP> <VARNAME> \n", progname);
    printf("     For example: %s 192.168.1.42 stat.time.TimeStr
        \n", progname);
    printf("\n");
  }

  int OnError(int errorcode, const char* description,
     HController HC)
  {
    printf("ErrorNr: %i, Description: %s\n", errorcode,
        description);

    return 0;
  }

  /**
   * Prints the value of a variable as string.
   */
  void printVariable(HController HC, const char* varname) {
    /* Get the Handle for a NodeObject */
    HNodeObject HNO = GetNode(HC, varname);

    ARG_NODEINFO *nodeinfo = GetNodeInfo(HC, HNO);
    if ( nodeinfo ) {
      printf("Fullname: %s\n", nodeinfo->fullname);
      printf("Lastname: %s\n", nodeinfo->name);
      printf("Value:    %s\n", nodeinfo->value);
      printf("Unit:     %s\n", nodeinfo->unit);
      printf("Type:     %s\n", nodeinfo->typestring);
      DestroyNodeInfo(nodeinfo);
    } else {
      printf("Node %s not found.\n",varname);
```

```
  }
}

int main(int argc, char *argv[])
{
  char *ip;                    /* IP-Address of the ASC-
     Controller */
  short port;                  /* Port of the ASC-Controller */
  char *varname;

  HController HC;              /* Handle to our ASC */

  if (argc != 3) {
    usage(argv[0]);
    exit(1);
  }

  ip = argv[1];
  port = DEFAULT_PORT;
  varname = argv[2];

  /* Initialize the Library */
  InitControllerLib();
  RegisterOnError(OnError);

  /* Connect to the controller */
  HC = DetectRemoteController(ip, port);

  if (HC != (DWORD)INVALID_HANDLE_VALUE) {
    EnableVariableCache(HC);
    /* Print the variable if the controller is connected */
    printVariable(HC, varname);
  } else {
    printf(”Controller at %s:%d not found.\n”, ip, port);
  }
```

```
  /* Deinitialize the library */
  UnregisterOnErrorSingle(OnError);
  DeinitControllerLib();
}
```

5.5.3. Writing a variable

Listing 5.6: Writing a variable example

```
/**
 * Connect to a controller and set the value of a variable.
 *
 * For example:
 *   ./writeVariable 192.168.1.42 sys.BoardName MyBoardName
 */

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include "arg_controllerlib.h"

#define DEFAULT_PORT 1610

/**
 * Print out usage information
 */
void usage(const char *progname)
{
  printf("\n");
  printf("Write a new value to a node\n");
  printf("\n");
  printf("Usage:\n");
  printf("  %s <IP> <VARNAME> <VALUE>\n", progname);
  printf("     For example: %s 192.168.1.42 sys.BoardName
     MyBoardName\n", progname);
  printf("\n");
```

```
}

int OnError(int errorcode, const char* description,
   HController HC)
{
  printf("ErrorNr: %i, Description: %s\n", errorcode,
     description);

  return 0;
}

/**
 * Prints the value of a variable as string.
 */
void writeVariable(HController HC, const char* varname, const
    char* value) {

  /* Get the Handle for a NodeObject */
  HNodeObject HNO = GetNode(HC, varname);
  int allowed;
  if ( ModifyAllowed(HC,HNO,&allowed) == E_OK ) {
    if ( allowed ) {
      if ( SetNodeValueString(HC, HNO, value) != E_OK ) {
        printf("Error setting %s\n",value);
      } else {
        if ( WriteNode(HC, HNO) != E_OK ) {
          printf("Error writing %s\n",value);
        }
      }
    } else {
      printf("You are not allowed to change the value of this
          variable\n");
    }
  } else {
    printf("Variable not found: %s\n", varname);
  }
```

```
}

int main(int argc, char *argv[])
{
  char *ip;                    /* IP-Address of the ASC-
     Controller */
  short port;                  /* Port of the ASC-Controller */
  char *varname;
  char *value;

  HController HC;              /* Handle to our ASC */

  if (argc != 4) {
    usage(argv[0]);
    exit(1);
  }

  ip = argv[1];
  port = DEFAULT_PORT;
  varname = argv[2];
  value = argv[3];

  /* Initialize the Library */
  InitControllerLib();
  RegisterOnError(OnError);

  /* Connect to the controller */
  HC = DetectRemoteController(ip, port);

  if (HC != (DWORD)INVALID_HANDLE_VALUE) {
    /* Write the variable if the controller is connected */
    EnableVariableCache(HC);
    writeVariable(HC, varname, value);
  } else {
    printf("Controller at %s:%d not found.\n", ip, port);
  }
```

```
    /* Deinitialize the library */
    UnregisterOnErrorSingle(OnError);
    DeinitControllerLib();
}
```

5.5.4. Watching a variable

Listing 5.7: Watching a variable example

```
/**
 * Watch a variable.
 *
 * For example:
 *   ./WatchVariable 192.168.1.42 job.Job
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include "arg_controllerlib.h"

#include <unistd.h>

#define DEFAULT_PORT 1610

/**
 * Print out usage information
 */
void usage(const char *progname)
{
    printf("\n");
    printf("Watch a Variable\n");
    printf("\n");
    printf("Usage:\n");
```

```
    printf("  %s <IP> <JOBNAME> \n", progname);
    printf("     For example: %s 192.168.1.42 stat.time.TimeStr
        \n", progname);
    printf("\n");
  }

  int OnError(int errorcode, const char* description,
      HController HC)
  {
    printf("ErrorNr: %i, Description: %s\n", errorcode,
        description);

    return 0;
  }

  int OnVariableChange(HController HC, HNodeObject HNO) {
    ARG_NODEINFO *info = GetNodeInfo(HC, HNO);
    if ( info != NULL ) {
      printf("Variable %s changed value to %s %s\n", info->
          fullname, info->value, info->unit);
      DestroyNodeInfo(info);
    } else {
      printf("GetNodeInfo Error\n");
    }

    return 0;
  }

  int main(int argc, char *argv[])
  {
    char *ip;                    /* IP-Address of the ASC-
        Controller */
    short port;                  /* Port of the ASC-Controller */
    const char *varname;

    HController HC;              /* Handle to our ASC */
```

```
HNodeObject HNO;          /* Handle to our NodeObject */

if (argc != 3) {
  usage(argv[0]);
  exit(1);
}

ip = argv[1];
port = DEFAULT_PORT;
varname = argv[2];

/* Initialize the Library */
InitControllerLib();
RegisterOnError(OnError);

/* Connect to the controller */
HC = DetectRemoteController(ip, port);

if (HC != (DWORD)INVALID_HANDLE_VALUE) {
  EnableVariableCache(HC);
  if ( (HNO = GetNode(HC, varname)) != INVALID_HANDLE_VALUE
      ) {
    OnVariableChange(HC, HNO);
    RegisterOnValueChanged(HC, varname, OnVariableChange);
  }
  while (true) {
    sleep(10);
  }
} else {
  printf("Controller at %s:%d not found.\n", ip, port);
}

/* Deinitialize the library */
UnregisterOnErrorSingle(OnError);
DeinitControllerLib();
```

```
}
```

5.5.5. Loading a job

Listing 5.8: Loading a job example

```
/**
 * Load a job to the controller.
 *
 * For example:
 *   ./LoadJob 192.168.1.42 job.Job
 */

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include "arg_controllerlib.h"

#define DEFAULT_PORT 1610

/**
 * Print out usage information
 */
void usage(const char *progname)
{
  printf("\n");
  printf("Load a Job to controller.\n");
  printf("\n");
  printf("Usage:\n");
  printf("  %s <IP> <JOBNAME> \n", progname);
  printf("     For example: %s 192.168.1.42 job.Job\n",
     progname);
  printf("\n");
}

/* Gets called, when an Error occurs */
```

```
int OnError(int errorcode, const char* description,
    HController HC) {
  printf("Error: %i, Description: %s\n", errorcode,
     description);
  return 0;
}

/**
 * Loads the Job to the controller
 */
void loadJob(HController HC, const char* jobname) {

  if ( LoadJob(HC, jobname, 1, 1) == E_OK ) {
    printf("Job loaded and selected\n");
  } else {
    printf("Failed to load Job %s\n",jobname);
  }
}

int main(int argc, char *argv[])
{
  char *ip;                  /* IP-Address of the ASC-
     Controller */
  short port;                /* Port of the ASC-Controller */
  char *jobname;

  HController HC;            /* Handle to our ASC */

  if (argc != 3) {
    usage(argv[0]);
    exit(1);
  }

  ip = argv[1];
  port = DEFAULT_PORT;
  jobname = argv[2];
```

```
    /* Initialize the Library */
    InitControllerLib();
    RegisterOnError(OnError);

    /* Connect to the controller */
    HC = DetectRemoteController(ip, port);

    if (HC != (DWORD)INVALID_HANDLE_VALUE) {
      SetPenPath(HC, "TestJobs/Pens");
      SetFontPath(HC, "TestJobs/Fonts");
      loadJob(HC, jobname);
    } else {
      printf("Controller at %s:%d not found.\n", ip, port);
    }

    /* Deinitialize the library */
    DeinitControllerLib();
  }
```

5.5.6. Marking a job

Listing 5.9: Marking a job example

```
/**
 * Load a job to the controller and start it.
 *
 * For example:
 *   ./MarkJob 192.168.1.42 job.Job
 */

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include "arg_controllerlib.h"
```

```
#include ”unistd.h”

#define PLC_DEVICES_FAILURE          2
#define PLC_DEVICES_READY            4
#define PLC_JOB_READY               16
#define PLC_JOB_ACTIVE              32

#define DEFAULT_PORT 1610

unsigned int g_PLCstate = 0;
unsigned int g_startjobwhenready = 0;
unsigned int g_jobactive = 0;
unsigned int g_jobdone = 0;
unsigned int g_canquit = 0;

/**
 * Print out usage information
 */
void usage(const char *progname)
{
  printf(”\n”);
  printf(”Load a Job to a controller and mark it.\n”);
  printf(”\n”);
  printf(”Usage:\n”);
  printf(”  %s <IP> <JOBNAME> \n”, progname);
  printf(”     For example: %s 192.168.1.42 job.Job\n”,
     progname);
  printf(”\n”);
}

/* Gets called, when an Error occurs */
int OnError(int errorcode, const char* description,
   HController HC) {
  printf(”Error: %i, Description: %s\n”, errorcode,
     description);
  return 0;
```

```
}

/**
 * Gets called when the plcstate changed
 */
int PLCChangeCallback(HController HC, unsigned int value,
    unsigned int reserved)
{
  g_PLCstate = value;

  if ( g_PLCstate & PLC_JOB_READY ) {
    if ( g_jobactive ) {
      printf("Job done\n");
      g_jobactive = 0;
      g_canquit = 1;
    } else if ( g_startjobwhenready ) {
      printf("Starting Job\n");
      JobStart(HC);
    }
  }
  if ( g_PLCstate & PLC_JOB_ACTIVE ) {
    printf("Job Active\n");
    g_jobactive=1;
  }

  return 0;
}

/**
 * Loads the Job to the controller
 */
void loadJob(HController HC, const char* jobname) {

  if ( LoadJob(HC, jobname, 1, 1) == E_OK ) {
    printf("Job loaded and selected\n");
  } else {
```

```
    printf(”Failed to load Job %s\n”,jobname);
  }
}

int main(int argc, char *argv[])
{
  char *ip;                     /* IP-Address of the ASC-
     Controller */
  short port;                   /* Port of the ASC-Controller */
  char *jobname;

  HController HC;               /* Handle to our ASC */

  if (argc != 3) {
    usage(argv[0]);
    exit(1);
  }

  ip = argv[1];
  port = DEFAULT_PORT;
  jobname = argv[2];

  /* Initialize the Library */
  InitControllerLib();
  RegisterOnError(OnError);

  /* Connect to the controller */
  HC = DetectRemoteController(ip, port);

  if (HC != (DWORD)INVALID_HANDLE_VALUE) {
    SetPenPath(HC, ”TestJobs/Pens”);
    SetFontPath(HC, ”TestJobs/Fonts”);
    if ( RegisterOnPLCChanged(HC, PLCChangeCallback) == E_OK
       ) {
      if ( g_PLCstate & PLC_JOB_ACTIVE ) {
        printf(”Another Job is already running!\n”);
```

```
        } else {
          printf("Loading Job\n");
          g_startjobwhenready = 1;
          loadJob(HC, jobname);
        }
      }

      // In a real program you would do this with signals
      while (!g_canquit) sleep(1);
    } else {
      printf("Controller at %s:%d not found.\n", ip, port);
    }

    /* Deinitialize the library */
    UnregisterOnError(OnError);
    DeinitControllerLib();
  }
```

A. Copyright and licenses

The ARGES ControllerLib is based in part on several open source software components the use of which is subject to the respective licenses.

The source code of the open source software will be sent to you and any third party on request on a data carrier, the production costs will be asserted in return. This offer is valid for three years after shipment of the product respectively after downloading the software. Please send your inquiry to info@arges.de.

Since the open source software is free software, the developers of this software exclude the liability. Please note that the warranty for the ARGES ControllerLib is not affected and fully covered.

The following sections list the copyright of the software components and relate the software components to the respective licenses.

A.1. Proprietary software

ControllerLib Copyright © 2007–2023 ARGES GmbH, Germany

Licensed under the InScript License on page 318

A.2. Open source software

FreeType Copyright © 1996–2002, 2006 David Turner, Robert Wilhelm, and Werner Lemberg

On 2019-10-23 the page https://www.freetype.org/contact.html further lists Alexei Podtelezhnikov and Suzuki Toshiya as core developers and Oran Agra, Graham Asher, David Bevan, Bradley Grainger, Infinality, Tom Kacvinsky, Pavel Kaňkovský, Antoine Leca, Just van Rossum, and Wu Chia-I as major contributors.

Licensed under the B.1 FreeType Project License

ImageMagick Copyright © 1999–2017 ImageMagick Studio LLC

Licensed under the B.2 ImageMagick License

TinyXML Copyright © Lee Thomason, Yves Berquin, Andrew Ellerton, the tinyXml community

Licensed under the B.4 Zlib License

B. License texts

The ARGES ControllerLib is based in part on several open source software components the use of which is subject to the respective licenses. The following sections list the license texts.

B.1. FreeType Project License

Retrieved 2019-10-23 from
https://git.savannah.gnu.org/cgit/freetype/freetype2.git/tree/docs/FTL.TXT

```
  This license  was inspired by the BSD,  Artistic, and IJG
  (Independent JPEG  Group) licenses, which  all encourage inclusion
  and  use of  free  software in  commercial  and freeware  products
  alike.  As a consequence, its main points are that:

    o We don’t promise that this software works. However, we will be
      interested in any kind of bug reports. (‘as is’ distribution)

    o You can  use this software for whatever you  want, in parts or
      full form, without having to pay us. (‘royalty-free’ usage)

    o You may not pretend that  you wrote this software.  If you use
      it, or  only parts of it,  in a program,  you must acknowledge
      somewhere  in  your  documentation  that  you  have  used  the
      FreeType code. (‘credits’)

  We  specifically  permit  and  encourage  the  inclusion  of  this
  software, with  or without modifications,  in commercial products.
  We  disclaim  all warranties  covering  The  FreeType Project  and
  assume no liability related to The FreeType Project.


  Finally,  many  people  asked  us  for  a  preferred  form  for  a
  credit/disclaimer to use in compliance with this license.  We thus
  encourage you to use the following text:

   „„„
    Portions of this software are copyright © <year> The FreeType
    Project (www.freetype.org).  All rights reserved.
   „„„

  Please replace <year> with the value from the FreeType version you
  actually use.


Legal Terms
===========

0. Definitions
--------------

  Throughout this license,  the terms ‘package’, ‘FreeType Project’,
  and  ‘FreeType  archive’ refer  to  the  set  of files  originally
  distributed  by the  authors  (David Turner,  Robert Wilhelm,  and
  Werner Lemberg) as the ‘FreeType Project’, be they named as alpha,
```

```
  beta or final release.

  ‘You’ refers to  the licensee, or person using  the project, where
  ‘using’ is a generic term including compiling the project’s source
  code as  well as linking it  to form a  ‘program’ or ‘executable’.
  This  program is  referred to  as  ‘a program  using the  FreeType
  engine’.

  This  license applies  to all  files distributed  in  the original
  FreeType  Project,   including  all  source   code,  binaries  and
  documentation,  unless  otherwise  stated   in  the  file  in  its
  original, unmodified form as  distributed in the original archive.
  If you are  unsure whether or not a particular  file is covered by
  this license, you must contact us to verify this.

  The FreeType  Project is copyright (C) 1996-2000  by David Turner,
  Robert Wilhelm, and Werner Lemberg.  All rights reserved except as
  specified below.

1. No Warranty
--------------

  THE FREETYPE PROJECT  IS PROVIDED ‘AS IS’ WITHOUT  WARRANTY OF ANY
  KIND, EITHER  EXPRESS OR IMPLIED,  INCLUDING, BUT NOT  LIMITED TO,
  WARRANTIES  OF  MERCHANTABILITY   AND  FITNESS  FOR  A  PARTICULAR
  PURPOSE.  IN NO EVENT WILL ANY OF THE AUTHORS OR COPYRIGHT HOLDERS
  BE LIABLE  FOR ANY DAMAGES CAUSED  BY THE USE OR  THE INABILITY TO
  USE, OF THE FREETYPE PROJECT.

2. Redistribution
-----------------

  This  license  grants  a  worldwide, royalty-free,  perpetual  and
  irrevocable right  and license to use,  execute, perform, compile,
  display,  copy,   create  derivative  works   of,  distribute  and
  sublicense the  FreeType Project (in  both source and  object code
  forms)  and  derivative works  thereof  for  any  purpose; and  to
  authorize others  to exercise  some or all  of the  rights granted
  herein, subject to the following conditions:

    o Redistribution of  source code  must retain this  license file
      (‘FTL.TXT’) unaltered; any  additions, deletions or changes to
      the original  files must be clearly  indicated in accompanying
      documentation.   The  copyright   notices  of  the  unaltered,
      original  files must  be  preserved in  all  copies of  source
```

```
      files.

    o Redistribution in binary form must provide a  disclaimer  that
      states  that  the software is based in part of the work of the
      FreeType Team,  in  the  distribution  documentation.  We also
      encourage you to put an URL to the FreeType web page  in  your
      documentation, though this isn’t mandatory.

  These conditions  apply to any  software derived from or  based on
  the FreeType Project,  not just the unmodified files.   If you use
  our work, you  must acknowledge us.  However, no  fee need be paid
  to us.

3. Advertising
--------------

  Neither the  FreeType authors and  contributors nor you  shall use
  the name of the  other for commercial, advertising, or promotional
  purposes without specific prior written permission.

  We suggest,  but do not require, that  you use one or  more of the
  following phrases to refer  to this software in your documentation
  or advertising  materials: ‘FreeType Project’,  ‘FreeType Engine’,
  ‘FreeType library’, or ‘FreeType Distribution’.

  As  you have  not signed  this license,  you are  not  required to
  accept  it.   However,  as  the FreeType  Project  is  copyrighted
  material, only  this license, or  another one contracted  with the
  authors, grants you  the right to use, distribute,  and modify it.
  Therefore,  by  using,  distributing,  or modifying  the  FreeType
  Project, you indicate that you understand and accept all the terms
  of this license.

4. Contacts
-----------

  There are two mailing lists related to FreeType:

    o freetype@nongnu.org

      Discusses general use and applications of FreeType, as well as
      future and  wanted additions to the  library and distribution.
      If  you are looking  for support,  start in  this list  if you
      haven’t found anything to help you in the documentation.
```

```
    o freetype-devel@nongnu.org

      Discusses bugs,  as well  as engine internals,  design issues,
      specific licenses, porting, etc.

  Our home page can be found at

    https://www.freetype.org


--- end of FTL.TXT ---
```

B.2. ImageMagick License

Retrieved 2019-10-23 from https://imagemagick.org/script/license.php

Terms and Conditions for Use, Reproduction, and Distribution

The legally binding and authoritative terms and conditions for use, reproduction, and distribution of ImageMagick follow:

Copyright © 1999–2017 ImageMagick Studio LLC, a non-profit organization dedicated to making software imaging solutions freely available.

1. Definitions.

License shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

Licensor shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

Legal Entity shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, control means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

You (or *Your*) shall mean an individual or Legal Entity exercising permissions granted by this License.

Source form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

Object form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

Work shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

Derivative Works shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

Contribution shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as *Not a Contribution*.

Contributor shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

a. You must give any other recipients of the Work or Derivative Works a copy of this License; and

b. You must cause any modified files to carry prominent notices stating that You changed the files; and

c. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

d. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an *AS IS* BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

B.3. InScript License

Retrieved 2019-10-23 from the InScript software

```
End user license agreement
====================================================================

Important - Read carefully!
----------------------------------
This End User License Agreement is a legal agreement between you
(either an individual or a single entity) and ARGES GmbH
(hereinafter ”ARGES”) for an ARGES software product, including
software, the associated media, any printed materials and any
”online” or electronic documentation (hereinafter generally
denominated ”software product”). By installing, copying or
otherwise using the software product, you agree to be bound by the
terms of this License Agreement.

Software product license
----------------------------------
The software product is protected by copyright laws and
international copyright treaties, as well as any other intellectual
property laws and treaties. The software product is licensed not
```

```
sold.


1. Grant of license
====================================================================

Software
----------------------------------
You may install and use one copy of the software product on a single
computer, or in place of a previous version destined for the same
operative system. The original user of the computer on which the
software is installed has the right to create a second copy for his
own exclusive use on a home computer or portable computer.

Storage/Network use
----------------------------------
You may also store or install a copy of the software product on a
storage medium, such as a network server, to the extent that such is
used exclusively for installing or using the software product on
your internal network. However, you must acquire and dedicate a
license for the software program for each computer on which the
software program is installed or to which it is distributed. A
license for the software program may not be shared or used
concurrently on different computers.

License package
----------------------------------
If you acquired the present License Agreement as part of an ARGES
license package you will be entitled to create other copies of the
software product in the authorized number according to the printed
copy of the present License Agreement. You can use each copy in the
manner indicated above. You also are entitled to produce a
corresponding number of copies for use on a home computer or
portable computer in conformity with the above mentioned clauses.


2. Description of other rights and limitations
====================================================================

Limitations on reverse engineering,
decompilation and disassembly
----------------------------------
You may not reverse engineer, decompile or disassemble the software
product, except and only to the extent that such activity is
expressly permitted by applicable law notwithstanding this
```

```
limitation.

Separation of components
----------------------------------
The software program is licensed as a single product. Its component
parts may not be separated for use on more than one computer.

Rental
----------------------------------
You may not rent or lease the software product.

Software transfer
----------------------------------
You may permanently transfer all of your rights under this Agreement
only as part of a sale or transfer, provided that you retain no
copies, transfer all of the software product (including all
component parts, the media and the printed materials, any upgrades,
The Agreement and, if applicable, the Certificate of Authenticity)
and the recipient agrees to the terms of this Agreement. If this
software product is an upgrade, any transfer must include all prior
versions of the software product.

Termination
----------------------------------
Without prejudice to other rights, ARGES may terminate this
Agreement if you fail to comply with the terms and conditions of the
Agreement. In such event, you must destroy all copies of the
software product and all of its component parts.


3. Upgrades
====================================================================

If the software product is an upgrade from another product, whether
from ARGES or another supplier, you may use or transfer the software
product only in conjunction with that upgraded product, unless you
destroy the upgraded product. If the software product is an upgrade
of an ARGES product, you may use that upgraded product only in
accordance with this Agreement. If the software product is an
upgrade of a component of a package of software programs which you
licensed as a single product, the software product may be used and
transferred only as part of that single product package and may not
be separated for use on more than one computer.
```

```
4. Copyright
====================================================================

All title and copyrights in and to the software product (including
but not limited to any images, photographs, animations, video,
audio, music, text and ”applets”, incorporated into the software
product), the accompanying printed material, and any copies of the
software product, are owned by ARGES. The software product is
protected by copyright laws and international treaty provisions.
Therefore you must treat the software product as any other type of
material covered by copyright, with the exclusion that you either
(a) create a single backup copy of the software product to be used
exclusively for backup and file requirements; or (b) install the
software product on a single computer if the original is kept for
backup and file requirements. You may not copy the printed materials
accompanying the software product.


5. Dual-media software
====================================================================

You may receive the software product in more than one medium.
Regardless of the type or size of the medium you receive, you may
only use one medium that is appropriate for your single computer.
You may not use or install the other medium on another computer.
You may not loan, rent, lease or otherwise transfer the other medium
to another user, except as part of a permanent transfer (as
provided above) of the software product.


6. Warranty conditions
====================================================================

Limits of the warranty
----------------------------------
ARGES guarantees for a period of 90 days from the date of delivery
to customer that (a) the medium/media on which the product is
supplied is free from material defects, (b) under normal use it
functions in conformity with that described in the accompanying
manual. ARGES supplies this guarantee as producer of the software.
The present provisions do not limit or substitute possible guarantee
rights or legal responsibility towards retailers from whom the user
purchases the software.

Customers’ rights
```

```
----------------------------------
In the case of guarantee claim the customer has the right, at the
discretion of ARGES, (a) to restitution of the price paid or (b)
substitution or supply of the missing parts of software which do not
form part of the ARGES guarantee provided that they are returned to
ARGES together with a copy of the purchase receipt. The present
guarantee is not applicable if the software fault was caused by
accident, improper use or erroneous application. Each substitute
software component is guaranteed by ARGES for the remaining period
of the original guarantee and in any case for not less than 30 days.

Warranty exclusions
----------------------------------
ARGES does not recognize any other warranty relative to software and
accompanying documentation (in printed or electronic form).

Exclusion of liability
for indirect damages
----------------------------------
ARGES or suppliers for its products will not be liable for damages
(including, without limitations, damages due to loss of income,
interruption of activity, loss of information or data, or other
economical damages) deriving from use of this product or from
inability to use the software, even in the case that ARGES is
advised of the possibility of such damages. In any case, the
liability of ARGES will be limited to an amount corresponding to
that actually paid for license concession. This exclusion of
responsibility is waived for damages caused by fraud or serious
fault on the part of ARGES. Rights relative to binding regulations
of responsibility for the product remain unprejudiced.

----------------------------------
2008-11-20

# end of file
```

B.4. Zlib License

Retrieved 2019-10-23 from https://directory.fsf.org/wiki/License:Zlib

```
This software is provided ’as-is’, without any express or implied
warranty. In no event will the authors be held liable for any damages
```

C. Trademarks

ARGES® is the registered trademark of the ARGES GmbH in Germany.

InScript® is the registered trademark of the ARGES GmbH in Germany.

Linux® is the registered trademark of Linus Torvalds in the U.S.A. and other countries.

macOS® is the registered trademark of Apple Inc. in the U.S.A. and other countries.

Qt® is the registered trademark of The Qt Company Ltd. in the U.S.A. and other countries.

Visual Studio® is a registered trademark of the Microsoft Corporation in the U.S.A. and other countries.

Windows® is the registered trademark of the Microsoft Corporation in the U.S.A. and other countries.

And just in case we missed some ...

All other trademarks cited herein are the property of their respective owners.

D. Bibliography

[1] *ARGES System Controller – Instruction Manual*.
See the file `ASC_controller_manual_en.pdf`.

[2] *Firmware 3 – User Manual*.
See further below at *Opening the InScript and Firmware User Manual on-screen*.

[3] *InScript 3 Software – User Manual*.
See further below at *Opening the InScript and Firmware User Manual on-screen* or see the file `InScript_<version>_manual_en.pdf`.

Opening the InScript and Firmware User Manuals on-screen

The InScript user manual consists of 2 parts.

The InScript software provides the first part, the *InScript User Manual*. This part explains how to use the InScript software.

The ARGES system controller provides the second part, the *Firmware User Manual*. This part covers controller-specific functionality as e.g. executable job nodes and supported devices.

As you can connect multiple ARGES system controllers to the InScript software, you must select the controller in question first in order to consult its controller-specific *Firmware User Manual*. When opening the InScript user manual on-screen the InScript software downloads the *Firmware User Manual* from the ASC system controller that is connected to the InScript software and selected in the *Navigator* view and merges the *InScript User Manual* and the *Firmware User Manual* to a single help file.

Requirements

- ☐ If you want to open the *InScript User Manual* and the *Firmware User Manual* of a specific ARGES system controller then the controller in question must be connected to the InScript software and selected in the *Navigator* view.

Procedure

- In the menu of the InScript software, click **Help** → **Manual**.

  The *Help* window opens and shows the *InScript User Manual* and the controller-specific *Firmware User Manual* as contents.

E. Acronyms

ARL ARGES Raw Lines

ARL is an efficient vector graphics file format specified by ARGES. It is the recommended file format for 2D and 3D vector data in InScript. If you want to create ARL files with your own software then please contact our service department to get the file format description

FDT is a font file format used by the ControllerLib

NVRAM Non-Volatile Random-Access Memory

PLC Programmable Logic Controller

TTF TrueType Font

Index

AcceptsSubnodes, 130
ActivateDevice, 180
ActivateDeviceByName, 181
AddNodeObject, 154
AddNodeObjectByName, 154
AddNodeObjectSubtreeByName, 155
ARG_DEVICEINFO, 17
ARG_DEVICESTATE, 17
ARG_DRIVERINFO, 16
ARG_Feature, 19
ARG_FeatureList, 19
ARG_JOBINFO, 14
ARG_JOBNODEINFO, 13
ARG_LINEINFO, 15
ARG_NODEINFO, 12
ARG_SINGLEDEVICEINFO, 18
ARG_SINGLEDRIVERINFO, 16
ARG_TSS_CHANNELTYPE, 19
ARG_TSS_Data, 20
ARG_TSS_INFO, 20

BeginSpecialJob, 172

CancelTssDataConnection, 172
CanMakeLines, 129
CanPassLines, 130
CLFree, 48
ControllerCount, 56
CreateAllowed, 118
CreateDevice, 176
CreateJobNodeOnController, 108
CreateJobNodeOnControllerAsync, 108
CreateJobNodeOnControllerExt, 107
CreateNodeObjectCollection, 153
CreateNodeOnController, 104
CreateNodeOnControllerAsync, 106
CreateNodeOnControllerExt, 105
CreatePenVar, 187

DeactivateDevice, 181
DeactivateDeviceByName, 182
DeinitControllerLib, 47
DeleteAllowed, 117
DeleteDevice, 177
DeleteDeviceByName, 177
DeleteNode, 100
DeleteNodeOnController, 100
DeleteNodeOnControllerByName, 101
DeleteSingleNodeOnController, 101
DeleteSingleNodeOnControllerByName, 102
DeselectNode, 97
DeselectNodeByName, 98
DestroyDeviceInfo, 179
DestroyDriverInfo, 175
DestroyFeatureList, 55
DestroyFontInfo, 71
DestroyJobInfo, 162
DestroyJobNodeTypeInfo, 166
DestroyNodeInfo, 89
DestroyNodeObjectCollection, 153
DestroySingleDeviceInfo, 180
DestroyTssInfo, 171
DetectRemoteController, 53
DeviceActivatedCallbackFunction, 37
DeviceCreatedCallbackFunction, 37
DeviceDeactivatedCallbackFunction, 37
DeviceDeletedCallbackFunction, 37
DeviceDependencyAddedCallbackFunction, 37
DeviceDependencyRemovedCallbackFunction, 38
DeviceErrorStateChangedCallbackFunction, 38

DeviceParamStateChangedCallbackFunction, 38
DevicePowerStateChangedCallbackFunction, 38
DeviceStateChangedCallbackFunction, 38
DisableFloatingPointToStringDot, 49
DisconnectController, 57

EnableVariableCache, 56
EnableWatchdog, 52
EndOfNodeCreatedRequestCallbackFunction, 26
EndOfNodeDeletedRequestCallbackFunction, 29
EndSpecialJob, 173
ErrorCallbackFunction, 39
ExecuteScript, 67
ExtendRangeAllowed, 119

FlagsChangeCallbackFunction, 23
FlagsChangeCallbackFunctionExt, 23
FlagsModifyAllowed, 118
FreeCorrectionGridData, 57
FreeQCHFile, 61
FreeQHCFile, 61
FreeRCCFile, 61
FreeSaveTreeString, 78

GetActiveScanfieldCorrection, 255
GetAllAvailableFontnames, 71
GetAllDeviceInfo, 178
GetApplicationName, 50
GetControllerLibVersionString, 48
GetControllerQCHFile, 59
GetControllerQHCFile, 60
GetControllerRCCFile, 58
GetCorrectionGrid, 57
GetDevice, 176
GetDeviceInfo, 179
GetDriverInfo, 174
GetEditorHint, 151
GetFeatureList, 54
GetFontPath, 70
GetHead, 253
GetHeadCount, 253
GetJobInfo, 161
GetJobLines, 166
GetJobLinesAbort, 167
GetJobNames, 163
GetJobNamesLen, 162
GetJobNodeTypeInfo, 165
GetJobNodeTypesCount, 164
GetLogLevel, 258
GetMax, 145
GetMin, 144
GetNode, 86
GetNodeFlags, 92
GetNodeFromCache, 86
GetNodeID, 113
GetNodeIndex, 109
GetNodeInfo, 87
GetNodeInfoExt, 88
GetNodeLastName, 115
GetNodeLastNameLen, 115
GetNodeName, 114
GetNodeNameLen, 113
GetNodeObjectAtIndex, 158
GetNodeObjectCount, 157
GetNodeState, 95
GetNodeType, 110
GetNodeTypeString, 112
GetNodeTypeStringLen, 111
GetNodeValueBinCopy, 134
GetNodeValueBinCopyLen, 134
GetNodeValueBool, 136
GetNodeValueInt32, 138
GetNodeValueInt64, 139
GetNodeValueReal32, 140
GetNodeValueReal64, 140
GetNodeValueString, 137
GetNodeValueStringLen, 136
GetParentNode, 95
GetPenPath, 69
GetPLCState, 64
GetSelectEntriesCount, 131
GetSelectEntry, 132
GetSelectEntryLength, 131
GetSubnodes, 155
GetSubnodesCount, 96
GetSubnodesHNOArray, 149
GetSubnodesNames, 148
GetSubnodesNamesLen, 147
GetSysMsgRawText, 250
GetSysMsgRawTextLen, 249
GetSysMsgXML, 246

GetSysMsgXMLLen, 245
GetSystemMessageXML, 242
GetSystemMessageXMLLen, 242
GetTssInfo, 170
GetUniqueHandle, 58
GetUnit, 146
GetUnitLen, 146

ImproveScanfieldCorrection, 255
InitControllerLib, 47
IsConsumable, 120
IsControlledByDevice, 122
IsDeviceResettable, 185
IsDeviceResettableByName, 186
IsForcePen, 127
IsFullFeatured, 51
IsInitialized, 48
IsManaged, 184
IsManagedByName, 185
IsMirrored, 124
IsModified, 124
IsNodeChildOf, 150
IsNodeDescendantOf, 150
IsOwnedByDevice, 123
IsPenable, 121
IsProtected, 123
IsQuicksaveable, 126
IsUserdefined, 128
IsWriteable, 128

JobAbort, 66
JobClearAll, 164
JobLinesCallbackFunction, 41
JobNodesOffset, 167
JobNodesRotate, 168
JobNodesScale, 168
JobNodesTransform, 169
JobPilot, 65
JobStart, 64
JobStop, 66

LoadDataXML, 84
LoadDataXMLFromFile, 84
LoadDistortion, 254
LoadFont, 83
LoadJob, 72
LoadJobExt, 73
LoadPen, 82
LoadTree, 79
LoadTreeFromString, 80
LoadTreeFromStringExt, 81
Log, 260
LogCallbackFunction, 40

ModifyAllowed, 116
ModifyMinMaxAllowed, 119
ModifyUnitAllowed, 120
MoveSubtree, 103

NameChangeCallbackFunction, 24
NameChangeCallbackFunctionExt, 24
NodeCreatedCallbackFunction, 25
NodeCreatedCallbackFunctionExt, 25
NodeDeletedCallbackFunction, 28
NodeDeletedCallbackFunctionExt, 27
NodeModifiedCallbackFunction, 21
NodeMovedCallbackFunction, 30
NodeMovedCallbackFunctionExt, 30
NodeStateChangeCallbackFunction, 31
NodeWillBeDeletedCallbackFunction, 29

PLCChangeCallbackFunction, 34
PLCChangeCallbackFunctionExt, 33
ProbeRemoteController, 53

ReadNode, 90
ReadNodeObjectCollection, 158
RegisterOnDeviceActivated, 230
RegisterOnDeviceCreated, 228
RegisterOnDeviceDeactivated, 231
RegisterOnDeviceDeleted, 229
RegisterOnDeviceDependencyAdded, 232
RegisterOnDeviceDependencyRemoved, 233
RegisterOnDeviceErrorStateChanged, 235
RegisterOnDeviceParamStateChanged, 236
RegisterOnDevicePowerStateChanged, 237
RegisterOnDeviceStateChanged, 234
RegisterOnEndOfNodeCreatedRequest, 205
RegisterOnEndOfNodeDeletedRequest, 211
RegisterOnError, 62
RegisterOnFlagsChanged, 197
RegisterOnFlagsChangedExt, 194

RegisterOnFlagsChangedGlobal, 196
RegisterOnJobLines, 238
RegisterOnLog, 258
RegisterOnNameChanged, 216
RegisterOnNameChangedExt, 214
RegisterOnNameChangedGlobal, 215
RegisterOnNodeCreated, 202
RegisterOnNodeCreatedExt, 201
RegisterOnNodeDeleted, 209
RegisterOnNodeDeletedExt, 206
RegisterOnNodeDeletedGlobal, 206
RegisterOnNodeModified, 188
RegisterOnNodeModifiedGlobal, 190
RegisterOnNodeMoved, 198
RegisterOnNodeMovedExt, 199
RegisterOnNodeStateChanged, 218
RegisterOnNodeStateChangedGlobal, 219
RegisterOnNodeWillDeletedGlobal, 212
RegisterOnPLCChanged, 222
RegisterOnPLCChangedExt, 221
RegisterOnRequestLog, 256
RegisterOnStartOfNodeCreatedRequest, 204
RegisterOnStartOfNodeDeletedRequest, 210
RegisterOnSysMsg, 225
RegisterOnSysMsgExt, 223
RegisterOnSystemMessage, 227
RegisterOnTssData, 239
RegisterOnUpdateTssChannels, 240
RegisterOnValueChanged, 193
RegisterOnValueChangedExt, 190
RegisterOnValueChangedGlobal, 191
RemoveNodeObject, 157
RenameAllowed, 117
RenameNode, 103
RequestLogCallbackFunction, 40
RequestTssDataConnection, 171
ResetDevice, 186
ResetDeviceByName, 187

SaveDistortion, 254
SaveJob, 74
SaveJobXML, 75
SavePen, 81
SaveTree, 76
SaveTreeAsString, 77
SaveTreeAsXMLString, 78
SaveTreeXML, 76
SelectNode, 96
SelectNodeByName, 97
SetApplicationName, 49
SetApplicationVersion, 49
SetAttributes, 51
SetConsumable, 121
SetControlledByDevice, 122
SetFontPath, 70
SetForcePen, 127
SetFullFeatured, 50
SetInternalAttributes, 51
SetLogFilename, 259
SetLogLevel, 257
SetManaged, 182
SetManagedByName, 183
SetMirrored, 125
SetNodeFlags, 93
SetNodeMinMax, 93
SetNodeUnit, 94
SetNodeValueBin, 133
SetNodeValueBool, 135
SetNodeValueInt32, 142
SetNodeValueInt64, 142
SetNodeValueReal32, 143
SetNodeValueReal64, 144
SetNodeValueString, 141
SetPenPath, 69
SetQuicksaveable, 126
SetRequestLogFilename, 257
SetUnmanaged, 183
SetUnmanagedByName, 184
StartOfNodeCreatedRequestCallbackFunction, 26
StartOfNodeDeletedRequestCallbackFunction, 28
StartTssRecorder, 260
StopTssRecorder, 260
SupportsXMLJobFormat, 54
SysMsgCallbackFunction, 36
SysMsgCallbackFunctionExt, 35
SystemMessageCallbackFunction, 35

TeachInSetCurrent, 98
TeachInSetCurrentByName, 99

UnregisterOnDeviceActivated, 231

UnregisterOnDeviceCreated, 229
UnregisterOnDeviceDeactivated, 232
UnregisterOnDeviceDeleted, 230
UnregisterOnDeviceDependencyAdded, 233
UnregisterOnDeviceDependencyRemoved, 234
UnregisterOnDeviceErrorStateChanged, 236
UnregisterOnDeviceParamStateChanged, 237
UnregisterOnDevicePowerStateChanged, 238
UnregisterOnDeviceStateChanged, 235
UnregisterOnEndOfNodeCreatedRequest, 205
UnregisterOnEndOfNodeDeletedRequest, 212
UnregisterOnError, 62
UnregisterOnErrorSingle, 63
UnregisterOnFlagsChanged, 197
UnregisterOnFlagsChangedGlobal, 196
UnregisterOnFlagsChangedSingle, 198
UnregisterOnFlagsChangedSingleExt, 195
UnregisterOnJobLines, 239
UnregisterOnJobLinesSingle, 238
UnregisterOnLog, 259
UnregisterOnNameChanged, 218
UnregisterOnNameChangedGlobal, 216
UnregisterOnNameChangedGlobalSingle, 216
UnregisterOnNameChangedSingle, 217
UnregisterOnNameChangedSingleExt, 214
UnregisterOnNodeCreated, 203
UnregisterOnNodeCreatedSingle, 203
UnregisterOnNodeCreatedSingleExt, 202
UnregisterOnNodeDeleted, 209
UnregisterOnNodeDeletedGlobal, 207
UnregisterOnNodeDeletedGlobalSingle, 208
UnregisterOnNodeDeletedSingle, 210
UnregisterOnNodeDeletedSingleExt, 208
UnregisterOnNodeModified, 189
UnregisterOnNodeModifiedGlobal, 190
UnregisterOnNodeModifiedSingle, 189
UnregisterOnNodeMoved, 199
UnregisterOnNodeMovedSingle, 200
UnregisterOnNodeMovedSingleExt, 201
UnregisterOnNodeStateChanged, 220
UnregisterOnNodeStateChangedGlobal, 219
UnregisterOnNodeStateChangedSingle, 220
UnregisterOnNodeWillDeletedGlobal, 213
UnregisterOnPLCChanged, 222
UnregisterOnPLCChangedSingle, 223
UnregisterOnPLCChangedSingleExt, 221
UnregisterOnRequestLog, 256
UnregisterOnStartOfNodeCreatedRequest, 204
UnregisterOnStartOfNodeDeletedRequest, 211
UnregisterOnSysMsg, 225
UnregisterOnSysMsgSingle, 226
UnregisterOnSysMsgSingleExt, 224
UnregisterOnSystemMessage, 227
UnregisterOnSystemMessageSingle, 228
UnregisterOnTssData, 240
UnregisterOnTssDataSingle, 240
UnregisterOnUpdateTssChannels, 241
UnregisterOnUpdateTssChannelsSingle, 241
UnregisterOnValueChanged, 193
UnregisterOnValueChangedGlobal, 192
UnregisterOnValueChangedSingle, 194
UnregisterOnValueChangedSingleExt, 192
UserCreatable, 129

ValueChangeCallbackFunction, 22
ValueChangeCallbackFunctionExt, 22

WriteNode, 90
WriteNodeAndStoreLocal, 91
WriteNodeObjectCollection, 159
WriteNodeObjectCollectionSync, 159
WriteNodeSync, 91
WriteToNVRAM, 67

Novanta Corporation

125 Middlesex Turnpike
Bedford, MA 01730, USA

Phone: +1-781-266-5800
Email: Photonics@Novanta.com
Website: www.NovantaPhotonics.com

ARGES ControllerLib 2023-09-18
User Manual (Original)
2023-09-18