

SM151E Software Manual

Windows GCS 2.0 DLL (PI_GCS2_DLL)

Release: 2.5.0 Date: 2016-02-26

Physik Instrumente (PI) GmbH & Co. KG is the owner of the following trademarks:
PI®, PIC®, PICMA®, PLine®, PiezoWalk®, NEXACT®, NEXLINE®, NanoCube®,
Picoactuator®, PInano®, PIMag®

The following designations are protected company names or registered trademarks of third parties:
Windows, LabVIEW

Software products that are provided by PI are subject to the General Software License Agreement of Physik Instrumente (PI) GmbH & Co. KG and may incorporate and/or make use of third-party software components.

For more information, please read the General Software License Agreement and the Third Party Software Note linked below.

[General Software License Agreement](#)

[Third Party Software Note](#)

PI owns the following patents or patent applications for the technology field Piezo Stepping Drive (PiezoWalk®, NEXACT®, NEXLINE®):
DE10148267B4, EP1267478B1, EP2209202B1, EP2209203B1, US6800984B2, DE4408618B4

PI owns the following patents or patent applications for the technology field Multilayer Piezo Actuators (PICMA®):
DE10021919C2, DE10234787C1, ZL03813218.4, EP1512183A2, JP4667863, US7449077B2

PI owns the following patents or patent applications for the technology field Ultrasonic Piezo Motors (PLine®):
Germany: DE102004024656A1, DE102004044184B4, DE102004059429B4,
DE102005010073A1, DE102005039357B4, DE102005039358A1, DE102006041017B4,
DE102008012992A1, DE102008023478A1, DE102008058484A1, DE102010022812A1,
DE102010047280A1, DE102010055848, DE102011075985A1, DE102011082200A1,
DE102011087542B3, DE102011087542B3, DE102011087801B4, DE102011108175,
DE102012201863B3, DE19522072C1, DE19938954A1
Europe: EP0789937B1EP1210759B1, EP1267425B1, EP1581992B1, EP1656705B1,
EP1747594B1, EP1812975B1, EP1861740B1, EP1915787B2, EP1938397B1, EP2095441B1,
EP2130236B1, EP2153476B1, EP2164120B1, EP2258004B1, EP2608286A2
USA: US2010/0013353A1, US5872418A, US6765335B2, US6806620B1, US6806620B1,
US7218031B2, US7598656B2, US7737605B2, US7795782B2, US7834518B2, US7973451B2,
US8253304B2, US8344592B2, US8482185B2
Japan: JP2011514131, JP2011522506, JP3804973B2, JP4377956, JP4435695, JP4477069,
JP4598128, JP4617359, JP4620115, JP4648391, JP4860862, JP4914895, JP2013539346
China: ZL200380108542.0, ZL200580015994.3, ZL200580029560.9, ZL200580036995.6,
ZL200680007223.4, ZL200680030007.1, ZL200680042853.5
International patent applications: WO2009059939A2, WO2010121594A1, WO2012048691A2,
WO2012113394A1, WO2012155903A1, WO2013034146A3, WO2013117189A2

PI owns the following patents or patent applications for the technology field Magnetic Direct Drives (PIMag®):
WO212146709A2, DE102012207082A1

PI owns the following patents or patent applications for the technology field Piezo Inertia Drives (PIShift, PiezoMike):
EP2590315A1, WO213064513A1, DE102011109590A1, WO2013020873A1

© 1999-2016 Physik Instrumente (PI) GmbH & Co. KG, Karlsruhe, Germany. The text, photographs and drawings in this manual are protected by copyright. With regard thereto, Physik Instrumente (PI) GmbH & Co. KG retains all the rights. Use of said text, photographs and drawings is permitted only in part and only upon citation of the source.

Document Number SM151E, BRo, Release 2.5.0
PIGCS_2_0_DLL_SM151E.doc

Subject to change without notice. This manual is superseded by any new release. The newest release is available for download from our website (www.pi.ws).

Table of Contents

| | | |
|-----------|--|------------|
| 1. | Introduction..... | 5 |
| 2. | Software Installation..... | 5 |
| 3. | General Information About PI DLLs | 6 |
| 3.1. | Threads | 6 |
| 3.2. | DLL Handling..... | 6 |
| 3.2.1. | Using a Static Import Library..... | 6 |
| 3.2.2. | Using a Module Definition File | 6 |
| 3.2.3. | Using Windows API Functions | 6 |
| 3.3. | Function Calls..... | 7 |
| 3.3.1. | Error Return..... | 7 |
| 3.3.2. | Axis Identifiers | 7 |
| 3.3.3. | Axis Parameters | 7 |
| 3.4. | Types Used in PI Software | 8 |
| 3.4.1. | Boolean Values | 8 |
| 3.4.2. | NULL Pointers | 8 |
| 3.4.3. | C-Strings | 8 |
| 4. | Communication Functions..... | 9 |
| 4.1. | Usage and Overview | 9 |
| 4.2. | Function Description | 11 |
| 4.3. | Interface Settings..... | 16 |
| 5. | Functions for Sending and Reading Strings..... | 17 |
| 5.1. | Overview | 18 |
| 5.2. | Function Description | 18 |
| 6. | Basic Functions for GCS Commands | 19 |
| 6.1. | Overview | 19 |
| 6.2. | Function Description | 31 |
| 7. | Functions for GCS Commands for Wave Generator and DDL | 105 |
| 7.1. | Functions Overview | 106 |
| 7.2. | Function Documentation | 107 |
| 8. | Functions for User-Defined Stages..... | 120 |
| 8.1. | Overview | 120 |
| 8.2. | Function Description | 121 |

8.3. Stage Databases 121

8.4. Troubleshooting..... 122

9. Error Codes 123

1. Introduction

The PI GCS2_library allows controlling one or more PI controllers connected to a host PC. The PI General Command Set (GCS) is the PI standard command set and ensures the compatibility between different PI controllers.

The library is available for the following operating systems:

- **Windows** Vista Service Pack 1 (32 bit, 64 bit), Windows 7 (32 bit, 64 bit), Windows 8 (32 bit, 64 bit) and Windows 10 (32 bit, 64 bit): PI GCS2 DLL
See Section 3.2 starting on p. 6 for more information about PI DLLs.
- **Linux** operating systems (kernel 2.6, GTK 2.0, glibc 2.4): libpi_pi_gcs2.so.x.x.x and libpi_pi_gcs2-x.x.x.a where x.x.x gives the version of the library

NOTES

This manual was originally written for the Windows version of the GCS library (DLL), and so the terminology used in this document is that common with Windows DLLs. Nevertheless this manual can also be used for the Linux versions of the GCS library because there is no difference in the functionality of the library functions between the individual operating systems.

See Section 3 starting on p. 6 for more information about PI DLLs.

There are various sample programs for different programming languages to be found in the \Sample directory of the CD of your controller.

2. Software Installation

To install the PI GCS 2 DLL on your host PC, follow the installation instructions for the PC software in the user manual of the controller or install the feature

“PI_Programming_Files_PI_GCS2_DLL_Setup.exe” located in the “SingleSetups” folder of the product CD. Afterwards, the required files will be located in
C:\Users\Public\PI\PI_Programming_Files_PI_GCS2_DLL.

Copy these files to the location where the source code of your application is built (.h. and .lib) and where the application is executed (.dll).

With some controllers, you can select a parameter set appropriate for your stage from a stage database, see PI_CST() (p. 34) and “Functions for User-Defined Stages” (p. 120).

NOTE

Stage database files (PIStages2.dat and PIMicosStages2.dat) are installed in the ...\\PI\\GcsTranslator directory. In that directory, also the PI_UserStages2.dat database will be located which is created automatically the first time the PI_qVST() or PI_CST() functions of the PI GCS2 DLL are used.

The location of the PI directory is that specified upon installation, usually in C:\\ProgramData. If this directory does not exist, the EXE file that needs the stage databases will look in its own directory. Note that in PIMikroMove, you can use the *Version Info* item in the controller menu or the *Search for controller software* item in the *Connections* menu to identify the GCSTranslator path.

PI is constantly improving the PC software. Always install the latest version of the PC software and the standard stage databases. Follow the update instructions for the PC software in the user manual of the controller. With Windows operating systems, use the PI Update Finder.

3. General Information About PI DLLs

The information below is valid for the DLL described in this manual as well as for the DLLs for many other PI products.

3.1. Threads

This DLL is not thread-safe. The function calls of the DLL are not synchronized and can be safely used only by one thread at a time.

3.2. DLL Handling

To get access to and use the DLL functions, the library must be included in your software project. There are a number of techniques supported by the Windows operating system and supplied by the different development systems. The following sections describe the methods which are most commonly used. For detailed information, consult the relevant documentation of the development environment being used. (It is possible to use the PI_GCS2_DLL.DLL in Delphi projects. Please see <http://www.drBob42.com/delphi/headconv.htm> for a detailed description of the steps necessary.)

3.2.1. Using a Static Import Library

The PI_GCS2_DLL.DLL module is accompanied by the PI_GCS2_DLL.LIB file. This is the static import library which can be used by the Microsoft Visual C++ system for 32-bit applications. In addition, other systems, like the National Instruments LabWindows CVI or Watcom C++ can handle, i.e. understand, the binary format of a VC++ static library. When the static library is used, the programmer must:

Use a header or source file in which the DLL functions are declared, as needed for the compiler. The declaration should take into account that these functions come from a "C-Language" Interface. When building a C++ program, the functions have to be declared with the attribute specifying that they are coming from a C environment. The VC++ compiler needs an extern "C" modifier. The declaration must also specify that these functions are to be called like standard Win-API functions. That means the VC++ compiler needs to see a WINAPI or __stdcall modifier in the declaration.

Add the static import library to the program project. This is needed by the linker and tells it that the functions are located in a DLL and that they are to be linked dynamically during program startup.

3.2.2. Using a Module Definition File

The module definition file is a standard element/resource of a 16- or 32-bit Windows application. Most IDEs (integrated development environments) support the use of module definition files. Besides specification of the module type and other parameters like stack size, function imports from DLLs can be declared. In some cases the IDE supports static import libraries. If that is the case, the IDE might not support the ability to declare DLL-imported functions in the module definition file. When a module definition file is used, the programmer must:

Use a header or source file where the DLL functions have to be declared, which is needed for the compiler. In the declaration should be taken into account that these function come from a "C-Language" Interface. When building a C++ program, the functions have to be declared with the attribute that they are coming from a C environment. The VC++ compiler needs an extern "C" modifier. The declaration also must be aware that these functions have to be called like standard Win-API functions. Therefore the VC++ compiler needs a WINAPI or __stdcall modifier in the declaration.

Modify the module definition file with an IMPORTS section. In this section, all functions used in the program must be named. Follow the syntax of the IMPORTS statement. Example:

```
IMPORTS
  PI_GCS2_DLL.PI_IsConnected
```

3.2.3. Using Windows API Functions

If the library is not to be loaded during program startup, it can sometimes be loaded during program execution using Windows API functions. The entry point for each desired function has to be obtained. The DLL linking/loading with API functions during program execution can always be done,

independent of the development system or files which have to be added to the project. When the DLL is loaded dynamically during program execution, the programmer has to:

Use a header or source file in which local or global pointers of a type appropriate for pointing to a function entry point are defined. This type could be defined in a typedef expression. In the following example, the type `FP_PI_IsConnected` is defined as a pointer to a function which has an `int` as argument and returns a `BOOL` value. Afterwards a variable of that type is defined.

```
typedef BOOL (WINAPI *FP_PI_IsConnected)( int );
FP_PI_IsConnected pPI_IsConnected;
```

Call the Win32-API `LoadLibrary()` function. The DLL must be loaded into the process address space of the application before access to the library functions is possible. This is why the `LoadLibrary()` function has to be called. The instance handle obtained has to be saved for use by the `GetProcAddress()` function. Example:

```
HINSTANCE hPI_DLL = LoadLibrary("PI_GCS2_DLL.DLL\0");
```

Call the Win32-API `GetProcAddress()` function for each desired DLL function. To call a library function, the entry point in the loaded module must be known. This address can be assigned to the appropriate function pointer using the `GetProcAddress()` function. Afterwards the pointer can be used to call the function. Example:

```
pPI_IsConnected = (FP_PI_IsConnected)GetProcAddress(hPI_DLL,"PI_IsConnected\0");
if (pPI_IsConnected == NULL)
{
    // do something, for example
    return FALSE;
}
BOOL bResult = (*pPI_IsConnected)(1); // call PI_IsConnected(1)
```

3.3. Function Calls

The first argument to most function calls is the ID of the selected controller.

3.3.1. Error Return

Almost all functions will return a boolean value of type `BOOL` (see “Boolean Values” (p. 8)). The result will be zero if the DLL finds errors in the command or cannot transmit it successfully, or if the DLL internal error status is non-zero for another reason. If the command is acceptable and transmission is successful, and if the library has controller error checking enabled (see **PI_SetErrorCheck()**), the return value will further reflect the error status of the controller immediately after the command was sent. **TRUE** indicates no error. To find out what went wrong when the call returns **FALSE**, call **PI_GetError()** to obtain the error code, and, if desired, translate it to the corresponding error message with **PI_TranslateError()**. The error codes and messages are listed in “Error Codes” (p. 123).

3.3.2. Axis Identifiers

Many commands accept one or more axis identifiers. If no axes are specified (either by giving an empty string or a **NULL** pointer) some commands will address all connected axes. Axes names are separated by a space “ ”.

3.3.3. Axis Parameters

Parameters for specified axes are stored in an array passed to the function. The parameter for the first axis is stored in `array[0]`, for the second axis in `array[1]`, and so on. So, if you call `PI_qPOS("1 2 n3", double pos[3])`, the position for '1' is in `pos[0]`, for '2' in `pos[1]` and for '3' in `pos[2]`. If you call `PI_MOV("1 3", double pos[2])` the target position for '1' is in `pos[0]` and for '3' in `pos[1]`.

If conflicting specifications are present, only the **last** occurrence is actually sent to the controller with its argument(s). Thus, if you call `PI_MOV("1 1 2", pos[3])` with `pos[3] = { 1.0, 2.0, 3.0 }`, '1' will move to 2.0 and '2' to 3.0. If you then call `PI_qPOS("1 1 2", pos[3])`, `pos[0]` and `pos[1]` will contain 2.0 as the position of '1'.

3.4. Types Used in PI Software

3.4.1. Boolean Values

The library uses the convention used in **Microsoft's C++ for boolean values**. If your compiler does not support this directly, it can be easily set up: Just add the following lines to a central header file of your project:

```
typedef int BOOL;  
#define TRUE 1  
#define FALSE 0
```

3.4.2. NULL Pointers

In the library and the documentation "null pointers" (pointers pointing nowhere) have the value **NULL**. This is defined in the windows environment. If your compiler does not know this, simply use:

```
#define NULL 0
```

3.4.3. C-Strings

The library uses the C convention to handle strings. Strings are stored as char arrays with '\0' as terminating delimiter. Thus, the "type" of a c-string is char*. Do not forget to provide enough memory for the final '\0'. If you declare:

```
char* text = "HELLO";
```

it will occupy 6 bytes in memory. To remind you of the zero at the end, the names of the corresponding variables start with "sz".

4. Communication Functions

4.1. Usage and Overview

To use the DLL and communicate with the controller, the DLL must be initialized with one of the "connect" functions:

- `PI_InterfaceSetupDlg()`
- `PI_ConnectNIgpib()`
- `PI_ConnectRS232()`
- `PI_ConnectDaisyChainDevice()`
- `PI_ConnectTCPIP()`
- `PI_ConnectTCPIPbyDescription()`
- `PI_ConnectUSB()`
- `PI_ConnectUSBWithBaudRate()`.

To allow the handling of multiple controllers, the open functions return a non-negative ID. This is a kind of index to an internal array storing the information for the (different) controllers. All other calls addressing the same controller have this ID as their first parameter. `PI_CloseConnection()` (p.31) will close the connection to the specified controller and free its system resources.

Daisy Chain

Note that before connecting a daisy chain device using the `PI_ConnectDaisyChainDevice()` function, the daisy chain port has to be opened using the `PI_OpenRS232DaisyChain()` or the `PI_OpenUSBdaisyChain` function, whichever is the appropriate one.

After the daisy chain port has been opened all controllers connected to this daisy chain port can be "opened" using `PI_ConnectDaisyChainDevice()`. A connection to a daisy chain device is closed using the `PI_CloseConnection()` function. To close the daisy chain port the `PI_CloseDaisyChain()` function has to be called.

For controllers whose address can be set with DIP switches: In a daisy-chain, connected via USB or via RS-232, there must be one controller with address 1. It is not required that this controller is directly connected to the host PC, i.e. this controller does not have to be the first controller of the daisy-chain. If there is no controller in a daisy-chain with address 1 an error message occurs when you try to setup a connection. All controllers in a daisy chain must be set to the same baud rate.

TCP/IP

Before connecting a device using the `PI_ConnectTCPIPbyDescription()` function, its description string should be queried by `PI_EnumerateTCPIPDevices()`.

USB

Before connecting a device using the `PI_ConnectUSB()` function, its description string should be queried by `PI_EnumerateUSB()`.

USB and Virtual COM Ports

The USB driver for some PI devices consists of two layers: The basic USB layer and a virtual COM port. The COM port is offered for convenience. There are legacy applications which can communicate with RS-232 to additional hardware (e.g. microscopy application where you can store command sequences that can be sent over RS-232). These applications can use the PI device with Windows's built in RS-232 support using the virtual COM port – even if the hardware connection is USB.

If you have more than one device connected to the PC, the COM port assignment is randomly chosen by the operating system. E.g. on one PC two devices may be available as COM3 and COM4, while they may be available as COM4 and COM7 on a another PC.

If you use the USB driver directly (i.e. the basic USB layer) you can connect to the device using its serial number.

The PI_GCS2_DLL supports both ways: Use either PI_ConnectRS232() with the number of the virtual COM port (baud rate doesn't matter, use 115200), or use PI_ConnectUSB() with the serial number of the device. You can also call PI_EnumerateUSB () to get a list of all connected PI devices and select the device from the list.

List of Communications Functions

| Function | Short Description | Page |
|---|--|------|
| BOOL PI_CancelConnect (int threadId) | Cancel connecting thread with given ID | 11 |
| void PI_CloseConnection (int ID) | Close connection to the controller | 11 |
| void PI_CloseDaisyChain (int iPortId) | Close connection to the daisy chain port | 11 |
| int PI_ConnectDaisyChainDevice (int iPortId, int iDeviceNumber) | Open a daisy chain device | 12 |
| int PI_ConnectNIgpib (int iChannelNr, int iDeviceAddress) | Open a connection from a National Instruments IEEE 488 board to the controller | 12 |
| int PI_ConnectRS232 (int iPortNumber, int iBaudRate) | Open an RS-232 ("COM") interface to a controller | 12 |
| int PI_ConnectRS232ByDevName (const char* szDevName, int BaudRate) | Open an RS-232 interface to a controller for Linux | 12 |
| int PI_ConnectTCPIP (const char* szHostname, int port) | Open a TCP/IP connection to the controller | 12 |
| int PI_ConnectTCPIPbyDescription (const char* szDescription) | Open a TCP/IP connection to the controller using one of the identification strings listed by PI_EnumerateTCPIPDevices() | 13 |
| int PI_ConnectUSB (const char* szDescription) | Open an USB connection to a controller using one of the identification strings listed by PI_EnumerateUSB() | 13 |
| int PI_ConnectUSBWithBaudRate (const char* szDescription, int iBaudRate) | Open an USB connection to a controller using one of the identification strings listed by PI_EnumerateUSB() | 13 |
| int PI_EnableTCPIPScan (int iMask) | Selects the network hardware type (e.g. Lantronix XPort or other Ethernet device) which is to be found by PI_EnumerateTCPIPDevices() | 13 |
| int PI_EnumerateTCPIPDevices (char* szBuffer, int iBufferSize, const char* szFilter) | Lists the identification strings of all controllers available in the network via TCP/IP | 13 |
| int PI_EnumerateUSB (char* szBuffer, int iBufferSize, const char* szFilter) | Lists the identification strings of all controllers available via USB interfaces | 14 |
| int PI_GetControllerID (int threadID) | Get ID of connected controller for given threadID | 14 |
| int PI_GetError (int ID) | Get error status of the DLL and, if clear, that of the controller | 14 |
| int PI_InterfaceSetupDlg (const char* szRegKeyName) | Open dialog to let user select the interface and create a new PI object | 14 |

| Function | Short Description | Page |
|--|---|------|
| BOOL PI_IsConnected (int ID) | Check if there is a controller with an ID of <i>ID</i> | 14 |
| BOOL PI_IsConnecting (int threadID, BOOL* bConnecting) | Check if thread with given ID is running trying to establish communication | 15 |
| int PI_OpenRS232DaisyChain (int iPortNumber, int iBaudRate, int* piNumberOfConnectedDaisyChainDevices, char* szDeviceIDNs, int iBufferSize) | Open a RS-232 ("COM") interface to a daisy chain and set the baud rate of the daisy chain master | 15 |
| int PI_OpenUSB DaisyChain (const char* szDescription, int* piNumberOfConnectedDaisyChainDevices, char* szDeviceIDNs, int iBufferSize) | Open a USB interface to a daisy chain | 15 |
| BOOL PI_SetErrorCheck (int ID, BOOL bErrorCheck) | Set error-check mode of the library | 15 |
| BOOL PI_TranslateError (int iErrorNumber, char* szErrorMessage, int iBufferSize) | Translate error number to error message | 16 |
| int PI_TryConnectRS232 (int port, int baudrate) | Starts background thread which tries to establish connection to controller with given RS-232 settings | 16 |
| int PI_TryConnectUSB (const char* szDescription) | Starts background thread which tries to establish connection to controller with given USB settings | 16 |

4.2. Function Description

BOOL **PI_CancelConnect** (int *threadID*)

Cancel connecting thread with given thread ID

Arguments:

Thread ID

Returns:

TRUE if thread was cancelled, **FALSE** if no thread with given ID was running

void **PI_CloseConnection** (int *ID*)

Close connection to the controller associated with *ID*. *ID* will not be valid after this call.

Arguments:

ID ID of controller, if *ID* is not valid nothing will happen.

void **PI_CloseDaisyChain** (int *iPortID*)

Close connection to the daisy chain port associated with *iPortID*. *iPortID* will not be valid after this call.

Note that if there are still some open connections to one or more daisy chain devices, these connections will be closed automatically.

Arguments:

iPortID ID of the daisy chain port, if *iPortID* is not valid nothing will happen.

int PI_ConnectDaisyChainDevice (int *iPortId*, int *iDeviceNumber*)

Open a daisy chain device. All future calls to control this device need the ID returned by this call. Note that before connecting a daisy chain device using the PI_ConnectDaisyChainDevice() function, the daisy chain port has to be opened using the PI_OpenRS232DaisyChain() or the PI_OpenUSBDAaisyChain() function, whichever is the appropriate one.

After the daisy chain port has been opened all controllers connected to this daisy chain port can be "opened" using PI_ConnectDaisyChainDevice(). A connection to a daisy chain device is closed using the PI_CloseConnection() function. To close the daisy chain port the PI_CloseDaisyChain() function has to be called. Closing the daisy chain port automatically closes all still opened daisy chain devices.

Arguments:

iPortId the ID of the daisy chain port. This ID is returned by PI_OpenRS232DaisyChain().

iDeviceNumber the number of the daisy chain device to use, is a value between 1 and the *piNumberOfConnectedDaisyChainDevices* value of the PI_OpenRS232DaisyChain() function.

Returns:

ID of new object, -1 if interface could not be opened or no controller is responding.

int PI_ConnectNIgpib (int *iBoard*, const int *iDeviceAddress*)

Open a connection from a National Instruments IEEE 488 board to the controller. All future calls to control this controller need the ID returned by this call.

Arguments:

iBoard number of board (check with NI installation software)

iDeviceAddress address of connected device

Returns:

ID of new object, -1 if interface could not be opened or no controller is responding.

int PI_ConnectRS232 (int *iPortNumber*, int *iBaudRate*)

Open an RS-232 ("COM") interface to a controller. The call also sets the baud rate on the controller side. All future calls to control this controller need the ID returned by this call.

Arguments:

iPortNumber COM port to use (e.g. 1 for "COM1")

iBaudRate to use

Returns:

ID of new object, -1 if interface could not be opened or no controller is responding.

int PI_ConnectRS232ByDevName (const char* *szDevName*, int *BaudRate*)

Open an RS-232 interface to a controller with Linux. The call also sets the baud rate on the controller side. All future calls to control this controller need the ID returned by this call.

Arguments:

szDevName device interface name for RS232 connection (Linux)

iBaudRate to use

Returns:

ID of new object, -1 if interface could not be opened or no controller is responding.

int PI_ConnectTCPIP (const char* *szHostname*, int *port*)

Open a TCP/IP connection to the controller. All future calls to control this controller need the ID returned by this call. Will fail if there is already a connection.

Communication cannot be maintained after the controller is power-cycled or rebooted. The connection must then be closed and reopened.

Arguments:

szHostname host name of the controller, can be the IP address, e.g. "192.168.1.1" (Leading zeros may cause problems)

port port to connect to. For controllers from PI, the port is always 50000.

Returns:

ID of new object, -1 if interface could not be opened or no controller is responding, or controller responds that it is already connected via TCP/IP.

int PI_ConnectTCPIPbyDescription (const char* szDescription)

Open a TCP/IP connection to the controller using one of the identification strings listed by PI_EnumerateTCPIPDevices(). All future calls to control this controller need the ID returned by this call. Will fail if there is already a connection.

Communication cannot be maintained after the controller is power-cycled or rebooted. The connection must then be closed and reopened.

Arguments:

szDescription the description of the controller returned by PI_EnumerateTCPIPDevice

Returns:

ID of new object, -1 if interface could not be opened or no controller is responding, or controller responds that it is already connected via TCP/IP.

int PI_ConnectUSB (const char* szDescription)

Open an USB connection to a controller using one of the identification strings listed by PI_EnumerateUSB(). All future calls to control this controller need the ID returned by this call. Will fail if there is already a connection.

Communication cannot be maintained after the controller is power-cycled or rebooted. The connection must then be closed and reopened.

Arguments:

szDescription the description of the controller returned by PI_EnumerateUSB

Returns:

ID of new object, -1 if interface could not be opened or no controller is responding, or the controller responds that it is already connected via USB.

int PI_ConnectUSBWithBaudRate (const char* szDescription,int iBaudRate)

Open an USB connection to a controller using one of the identification strings listed by PI_EnumerateUSB(). By specifying the baud rate, a connection using a different baudrate than the standard will be established more quickly. All future calls to control this controller need the ID returned by this call. Will fail if there is already a connection.

Communication cannot be maintained after the controller is power-cycled or rebooted. The connection must then be closed and reopened.

Arguments:

szDescription the description of the controller returned by PI_EnumerateUSB

iBaudRate: to use

Returns:

ID of new object, -1 if interface could not be opened or no controller is responding, or the controller responds that it is already connected via USB.

int PI_EnableTCPIPScan (int iMask)

Selects the network hardware type (e.g. Lantronix XPort or other Ethernet device) which is to be found by PI_EnumerateTCPIPDevices(). By default, all devices will be found—change the settings only in special cases and if you know which Ethernet hardware is implemented in your controller.

Arguments:

iMask Bit mask 1 = UDP; 2 = XPORT

Returns:

The previous bit mask

int PI_EnumerateTCPIPDevices (char* szBuffer, int iBufferSize, const char* szFilter)

Lists the identification strings of all controllers available in the network via TCP/IP. Using the mask, you can filter the results for certain text.

Arguments:

szBuffer buffer for the TCP/IP devices description.

iBufferSize size of the buffer

szFilter only controllers whose descriptions match the filter are returned in the buffer (e.g. a filter of "E-517" will only return the E-517 controllers, and not all PI controllers).

Returns:

>= 0: the number of controllers in the list
<0: Error code

int PI_EnumerateUSB (char* szBuffer, int iBufferSize, const char* szFilter)

Lists the identification strings of all controllers available via USB interfaces. Using the mask, you can filter the results for certain text.

Arguments:

szBuffer buffer for the USB devices description.

iBufferSize size of the buffer

szFilter only controllers whose descriptions match the filter are returned in the buffer (e.g. a filter of "E-861" will only return the E-861 controllers, and not all PI controllers).

Returns:

>= 0: the number of controllers in the list
<0: Error code

int PI_GetControllerID(int threadID)

Get ID of connected controller for given thread ID.

Arguments:

Thread ID

Returns:

ID of new controller (>=0), error code (<0) if there was an error, no thread running, or thread has not finished yet

int PI_GetError (int ID)

Get error status of the DLL and, if clear, that of the controller. If the library shows an error condition, its code is returned, if not, the controller error code is checked using **PI_qERR()** (p.123) and returned. After this call the DLL internal error state will be cleared; the controller error state will be cleared if it was queried.

Returns:

error ID, see **Error codes** (p. 123) for the meaning of the codes.

int PI_InterfaceSetupDlg (const char* szRegKeyName)

Open dialog to let user select the interface and create a new PI object. All future calls to control this controller need the ID returned by this call. See **Interface Settings** (p. 16) for a detailed description of the dialogs shown.

Arguments:

szRegKeyName key in the Windows registry in which to store the settings, the key used is "HKEY_LOCAL_MACHINE\SOFTWARE\<your keyname>" if *keyname* is **NULL** or "" the default key "HKEY_LOCAL_MACHINE\SOFTWARE\PI\PI_GCS2_DLL" is used.

Note:

If your programming language is C or C++, use '\\' if you want to create a key and a subkey at once. To create "MyCompany\PI_GCS2_DLL" you must call
PI_InterfaceSetupDlg("MyCompany\\PI_GCS2_DLL")

Returns:

ID of new object, -1 if user pressed "CANCEL", the interface could not be opened, or no controller is responding.

BOOL PI_IsConnected (int ID)

Check if there is a controller with an ID of *ID*.

Returns:

TRUE if *ID* points to an existing controller, **FALSE** otherwise.

BOOL PI_IsConnecting(int *threadID*, BOOL* *bConnecting*)

Check if thread with given ID is running trying to establish communication.

Arguments:

Thread ID

bConnecting **TRUE** if thread is running **FALSE** if no thread is running with given ID

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

int PI_OpenRS232DaisyChain (int *iPortNumber*, int *iBaudRate*, int* *piNumberOfConnectedDaisyChainDevices*, char* *szDeviceIDNs*, int *iBufferSize*)

Open a RS-232 ("COM") interface to a daisy chain and set the baud rate of the daisy chain master. Note that calling this function does not open a daisy chain device—to get access to a daisy chain device you have to call `PI_ConnectDaisyChainDevice()`! All future calls to `PI_ConnectDaisyChain()` need the ID returned by `PI_OpenRS232DaisyChain()`. The *iDeviceNumber* of the `PI_ConnectDaisyChain()` function is a value between 1 and the *piNumberOfConnectedDaisyChainDevices*.

Arguments:

iPortNumber COM port to use (e.g. 1 for "COM1")

iBaudRate to use

piNumberOfConnectedDaisyChainDevices variable to receive the number of connected daisy chain devices.

szDeviceIDNs buffer to receive the IDN strings of the controllers (see `PI_qIDN()`).

iBufferSize the size of the buffer *szDeviceIDNs*.

Returns:

ID of new object, -1 if interface could not be opened or no controller is responding.

int PI_OpenUSBDAISYChain (const char* *szDescription*, long* *pNumberOfConnectedDaisyChainDevices*, char* *szDeviceIDNs*, int *iBufferSize*)

Open a USB interface to a daisy chain. Note that calling this function does not open a daisy chain device—to get access to a daisy chain device you have to call `PI_ConnectDaisyChainDevice()`! All future calls to `PI_ConnectDaisyChain()` need the ID returned by `PI_OpenUSBDAISYChain()`. The *iDeviceNumber* of the `PI_ConnectDaisyChain()` function is a value between 1 and the *piNumberOfConnectedDaisyChainDevices*.

Arguments:

szDescription the description of the controller returned by `PI_EnumerateUSB`

piNumberOfConnectedDaisyChainDevices variable to receive the number of connected daisy chain devices.

szDeviceIDNs buffer to receive the IDN strings of the controllers (see `PI_qIDN()`).

iBufferSize the size of the buffer *szDeviceIDNs*.

Returns:

ID of new object, -1 if interface could not be opened or no controller is responding.

BOOL PI_SetErrorCheck (int *ID*, BOOL *bErrorCheck*)

Set error-check mode of the library. With this call you can specify whether the library should check the error state of the controller (with "ERR?") after sending a command. This will slow down communications, so if you need a high data rate, switch off error checking and call `PI_GetError()` yourself when there is time to do so. You might want to use permanent error checking to debug your application and switch it off for normal operation. At startup of the library error checking is switched on.

Arguments:

ID ID of controller

bErrorCheck switch error checking on (**TRUE**) or off (**FALSE**)

Returns:

the old state, before this call

BOOL PI_TranslateError (int *iErrorNumber*, char* *szErrorMessage*, int *iBufferSize*)

Translate error number to error message.

Arguments:

iErrorNumber number of error, as returned from **PI_GetError()**.

szErrorMessage pointer to buffer that will store the message

iBufferSize size of the buffer

Returns:

TRUE if successful, **FALSE**, if the buffer was too small to store the message

Int PI_TryConnectRS232 (int *port*, int *baudrate*)

Starts background thread which tries to establish connection to controller with given RS-232 settings.

Arguments:

port COM port to use (e.g. 1 for "COM1")

baudrate to use

Returns:

ID of new thread (≥ 0), **error code** (< 0) if there was an error

Int PI_TryConnectUSB (const char* *szDescription*)

Starts background thread which tries to establish connection to controller with given USB settings.

Arguments:

szDescription the description of the controller returned by **PI_EnumerateUSB**

Returns:

ID of new thread (≥ 0), **error code** (< 0) if there was an error

4.3. Interface Settings

With **PI_InterfaceSetupDlg()**, p. 14, the *Connect* dialog is called. This dialog offers interface tab cards where you can configure and establish the connection (see descriptions below). Note that not all of the interfaces shown via the tab cards may be present on your controller.

RS-232

- **COM Port:** Select the desired COM port of the PC, something like "COM1" or "COM2". Only the ports available on the system are displayed.
- **Baud Rate:** The baud rate of the interface. The baud rate chosen will be set on both the host PC and the controller side of the interface.

USB

- Use the "Rescan" button to obtain all controllers available via USB. In the resulting list, click on the controller to which you want to connect. Use the "Serial Settings" button to specify the baudrate set with the DIP switches on the controller.

IEEE 488

- **Board ID:** ID of the National Instruments board installed (currently only National Instruments IEEE boards are supported). If only one board is installed this will be 0, as in the most cases. Use the National Instruments setup and test software to determine the board ID.

- Device Address: The address of the connected device. Please read the documentation of the connected device to determine its address setting and, if necessary, how to change it. The settings here and at the device must match.

TCP/IP

- Use the "Search for controllers" button to obtain all available controllers with their IP address and port settings. In the resulting list, click on the controller to which you want to connect and check that its IP address and port number are correctly transferred to the Hostname / TCP/IP Address and Port fields above the list.
- In the list, you can also identify the controllers which have already a TCP/IP connection open. If you try to connect to such a controller, an error message will arise, and no connection is possible.

5. Functions for Sending and Reading Strings

With PI library functions for GCS query commands the controller automatically continues processing subsequent functions only after the controller has retrieved the complete response from the input buffer.

This is valid for all query functions except if a query is sent as a string using `PI_Gcs_Commandset()`.

PI library functions for GCS commands are described in "Basic Functions for GCS Commands" (p. 19) and "Functions for GCS Commands for Wave Generator and DDL" (p. 105).

Example for a query function not using a string:

```
PI_qMOV (ID,"1",pdValue)
```

CAUTION

If a query command is sent as string using `PI_Gcs_Commandset()` it is necessary to make sure that the size of the response string matches the size of the input buffer. Otherwise it may happen that a response has not yet been retrieved completely before a next function is processed.

Therefore, if a query command is sent as string, it is necessary to query the size of the response string in the input buffer by sending `PI_GcsGetAnswerSize()` and to retrieve the response from input buffer by sending `PI_GcsGetAnswer()`.

The response to `PI_GcsGetAnswerSize()` determines the size (i.e., *iBufferSize*) that the input buffer (i.e., *szAxes*) must have to obtain the complete response to the query.

In some cases it can be necessary to query `PI_GetAnswerSize()` again after that, for it may take some time until the controller has delivered the complete response string. Then, it is recommended to keep querying `PI_GetAnswerSize()` until 0 is returned.

Example for a query command sent as a string:

```
PI_GcsCommandset (ID, "MOV? 1")
```

```
PI_GcsGetAnswerSize()
```

```
PI_GcsGetAnswer()
```

5.1. Overview

```
BOOL PI_GcsCommandset (int ID, const char* szCommand)  
BOOL PI_GcsGetAnswer (int ID, char* szAnswer, int iBufferSize)  
BOOL PI_GcsGetAnswerSize (int ID, int* piAnswerSize)
```

5.2. Function Description

BOOL PI_GcsCommandset (int *ID*, const char* *szCommand*)

Sends a GCS command to the controller. Any GCS command can be sent, but this command is intended to allow use of commands not having a function in the current version of the library.

See the User Manual of the controller for a description of the GCS commands which are understood by the controller firmware, for a command reference and for any limitations regarding the arguments of the commands.

Arguments:

ID ID of controller

szCommand the GCS command as string

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_GcsGetAnswer (int *ID*, char* *szAnswer*, int *iBufferSize*)

Gets the answer to a GCS command, provided its length does not exceed *bufsize*. The answers to a GCS command are stored inside the DLL, where as much space as necessary is obtained. Each call to this function returns and deletes the oldest answer in the DLL.

See the User Manual of the controller for a description of the GCS commands which are understood by the controller firmware, for a command reference and for any limitations regarding the arguments of the commands.

Arguments:

ID ID of controller

szAnswer the buffer to receive the answer.

iBufferSize the size of *szAnswer*.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_GcsGetAnswerSize (int *ID*, int* *piAnswerSize*)

Gets the size of an answer of a GCS command.

Arguments:

ID ID of controller

piAnswerSize pointer to integer to receive the size of the oldest answer waiting in the DLL.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

6. Basic Functions for GCS Commands

The functions listed in this chapter are based on the commands of the GCS. You can use a function only if the corresponding command is supported by your controller. See the user manual of the controller for the supported commands.

For all details regarding the functionality and arguments of commands, see the command descriptions in the user manual of the controller.

NOTE

If a query command is sent as string using `PI_Gcs_Commandset()` it is necessary to make sure that the size of the response string matches the size of the input buffer.

Otherwise it may happen that a response has not yet been retrieved completely before a next function is processed.

See "Functions for Sending and Reading Strings" (p. 17) for details.

6.1. Overview

| Function | Short Description | Page |
|--|--|------|
| BOOL PI_AAP (int <i>ID</i> , const char* <i>szAxis1</i> , double <i>dLength1</i> , const char* <i>szAxis2</i> , double <i>dLength2</i> , double <i>dAlignStep</i> , int <i>iNrRepeatedPositions</i> , int <i>iAnalogInput</i>) | Automated Alignment Part | 31 |
| BOOL PI_ACC (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Set Closed-Loop Acceleration | 31 |
| BOOL PI_ADD (int <i>ID</i> , const char* <i>szVariable</i> , double <i>value1</i> , double <i>value2</i>) | Add two values and save the result to a variable | 31 |
| BOOL PI_AOS (int <i>ID</i> , const char* <i>szAxes</i> , const double* <i>pdValueArray</i>) | Set an offset to the analog input for the given axis | 32 |
| BOOL PI_ATC (int <i>ID</i> , const int* <i>piChannels</i> , const int* <i>piValueArray</i> , int <i>iArraySize</i>) | Automatic calibration | 32 |
| BOOL PI_ATZ (int <i>ID</i> , const char* <i>szAxes</i> , const double* <i>pdLowVoltageArray</i> , const BOOL* <i>pbUseDefaultArray</i>) | Automatic zero-point calibration | 32 |
| BOOL PI_BRA (const int <i>ID</i> , char *const <i>szAxes</i> , BOOL * <i>pbValarray</i>) | Set brake on/off | 33 |
| BOOL PI_CCL (int <i>ID</i> , int <i>iCommandLevel</i> , const char* <i>szPassWord</i>) | Set command level of the controller | 33 |
| BOOL PI_CMO (int <i>ID</i> , const char* <i>szAxes</i> , const int* <i>piValueArray</i>) | Select closed-loop control mode | 33 |
| BOOL PI_CPY (int <i>ID</i> , const char* <i>szVariable</i> , const char* <i>szCommand</i>) | Copy a command response into a variable | 33 |
| BOOL PI_CST (int <i>ID</i> , const char* <i>szAxes</i> , const char* <i>szNames</i>) | Loads stage parameter values from a stage database | 34 |
| BOOL PI_CTI (int <i>ID</i> , const int* <i>piTriggerInputIds</i> , const int* <i>piTriggerParameterArray</i> , const char* <i>szValueArray</i> , int <i>iArraySize</i>) | Configures the trigger input conditions | 34 |
| BOOL PI_CTO (int <i>ID</i> , const int* <i>piTriggerOutputIds</i> , const int* <i>piTriggerParameterArray</i> , const double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Configures the trigger output conditions | 34 |

| Function | Short Description | Page |
|---|--|------|
| BOOL PI_CTOSTring (int <i>ID</i> , const int* <i>piTriggerOutputIds</i> , const int* <i>piTriggerParameterArray</i> , const char* <i>szValueArray</i> , int <i>iArraySize</i>) | Configures the trigger output conditions | 35 |
| BOOL PI_CTR (int <i>ID</i> , const char* <i>szAxes</i> , const double* <i>pdValueArray</i>) | Set target relative to current closed-loop target | 35 |
| BOOL PI_CTV (int <i>ID</i> , const char* <i>szAxes</i> , const double* <i>pdValueArray</i>) | Set absolute closed-loop target | 35 |
| BOOL PI_DCO (int <i>ID</i> , const char* <i>szAxes</i> , const BOOL* <i>pbValueArray</i>) | Sets drift compensation mode for given axes | 36 |
| BOOL PI_DEC (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Set Closed-Loop Deceleration | 36 |
| BOOL PI_DEL (int <i>ID</i> , int <i>iMilliseconds</i>) | Delay The Command Interpreter | 36 |
| BOOL PI_DFH (int <i>ID</i> , const char* <i>szAxes</i>) | Defines the current position as the axis home position | 36 |
| BOOL PI_DIO (int <i>ID</i> , const int* <i>piChannelsArray</i> , const BOOL* <i>pbValueArray</i> , int <i>iArraySize</i>) | Set Digital Output Lines | 37 |
| BOOL PI_DPA (int <i>ID</i> , const char* <i>szPassword</i> , const char* <i>szAxes</i> , const unsigned int* <i>iParameterArray</i>) | Resets parameters or settings to default values | 37 |
| BOOL PI_DRC (int <i>ID</i> , const int* <i>piRecordTableIdsArray</i> , const char* <i>szRecordSource</i> , const int* <i>piRecordOptionsArray</i>) | Set Data Recorder Configuration | 37 |
| BOOL PI_DRT (int <i>ID</i> , const int* <i>piRecordTableIdsArray</i> , const int* <i>piTriggerSourceArray</i> , const char* <i>szValues</i> , int <i>iArraySize</i>) | Set Data Recorder Trigger Source | 37 |
| BOOL PI_EAX (int <i>ID</i> , const char* <i>szAxes</i> , const BOOL* <i>pbValueArray</i>) | Enable Axis | 38 |
| BOOL PI_FDG (int <i>ID</i> , const char* <i>szScanRoutineName</i> , const char* <i>szScanAxis</i> , const char* <i>szStepAxis</i> , const char* <i>szParameters</i>) | Fast alignment: Defines a fast alignment gradient search routine | 38 |
| BOOL PI_FDR (int <i>ID</i> , const char* <i>szScanRoutineName</i> , const char* <i>szScanAxis</i> , const double <i>dScanAxisRange</i> , const char* <i>szStepAxis</i> , const double <i>dStepAxisRange</i> , const char* <i>szParameters</i>) | Fast alignment: Defines a fast alignment area scan routine. | 39 |
| BOOL PI_FED (int <i>ID</i> , const char* <i>szAxes</i> , const int* <i>piEdgeArray</i> , const int* <i>piParamArray</i>) | Find Edge | 40 |
| BOOL PI_FGC (int <i>ID</i> , const char* <i>szProcessIds</i> , const double* <i>pdScanAxisCenterValueArray</i> , const double* <i>pdStepAxisCenterValueArray</i>) | Fast alignment: Change center position of gradient search routine. | 40 |
| BOOL PI_FIO (int <i>ID</i> , const char* <i>szAxis1</i> , double <i>dLength1</i> , const char* <i>szAxis2</i> , double <i>dLength2</i> , double <i>dThreshold</i> , double <i>dLinearStep</i> , double <i>dAngleScan</i> , int <i>iAnalogInput</i>) | Fast Input-Output Alignment Procedure | 41 |
| BOOL PI_FLM (int <i>ID</i> , const char* <i>szAxis</i> , double <i>dLength</i> , double <i>dThreshold</i> , int <i>iAnalogInput</i> , int <i>iDirection</i>) | Fast Line Scan to Maximum | 41 |
| BOOL PI_FLS (int <i>ID</i> , const char* <i>szAxis</i> , double <i>dLength</i> , double <i>dThreshold</i> , int <i>iAnalogInput</i> , int <i>iDirection</i>) | Fast Line Scan | 41 |
| BOOL PI_FNL (int <i>ID</i> , const char* <i>szAxes</i>) | Fast Move To Negative Limit | 42 |
| BOOL PI_FPH (int <i>ID</i> , const char* <i>szAxes</i>) | Starts Phase Finding Process | 42 |
| BOOL PI_FPL (int <i>ID</i> , const char* <i>szAxes</i>) | Fast Move To Positive Limit | 42 |

| Function | Short Description | Page |
|---|--|------|
| BOOL PI_FRC (int <i>ID</i> , const char* <i>szProcessIdBase</i> , const char* <i>szProcessIdsCouplet</i>) | Fast alignment: Couples fast alignment routines to each other. | 42 |
| BOOL PI_FRF (int <i>ID</i> , const char* <i>szAxes</i>) | Fast Move To Reference Switch | 43 |
| BOOL PI_FRP (int <i>ID</i> , const char* <i>szScanRoutineNames</i> , const int* <i>piOptionsArray</i>) | Fast alignment: Stops, pauses or resumes a fast alignment routine. | 43 |
| BOOL PI_FRS (int <i>ID</i> , const char* <i>szScanRoutineNames</i>) | Fast alignment: Starts a fast alignment routine. | 43 |
| BOOL PI_FSA (int <i>ID</i> , const char* <i>szAxis1</i> , double <i>dLength1</i> , const char* <i>szAxis2</i> , double <i>dLength2</i> , double <i>dThreshold</i> , double <i>dDistance</i> , double <i>dAlignStep</i> , int <i>iAnalogInput</i>) | Fast Scan with Automated Alignment | 43 |
| BOOL PI_FSC (int <i>ID</i> , const char* <i>szAxis1</i> , double <i>dLength1</i> , const char* <i>szAxis2</i> , double <i>dLength2</i> , double <i>dThreshold</i> , double <i>dDistance</i> , int <i>iAnalogInput</i>) | Fast Scan with Abort | 44 |
| BOOL PI_FSM (int <i>ID</i> , const char* <i>szAxis1</i> , double <i>dLength1</i> , const char* <i>szAxis2</i> , double <i>dLength2</i> , double <i>dThreshold</i> , double <i>dDistance</i> , int <i>iAnalogInput</i>) | Fast Scan to Maximum | 44 |
| BOOL PI_GetAsyncBuffer (int <i>ID</i> , double ** <i>pnValArray</i>) | Get address of internal buffer | 45 |
| int PI_GetAsyncBufferIndex (int <i>ID</i>) | Get index used for the internal buffer | 45 |
| BOOL PI_GetDynamicMoveBufferSize (int <i>ID</i> , long* <i>pnSize</i>) | Get Memory Space For Trajectory Points | 45 |
| BOOL PI_GOH (int <i>ID</i> , const char* <i>szAxes</i>) | Go To Home Position | 45 |
| BOOL PI_HasPosChanged (int <i>ID</i> , const char* <i>szAxes</i> , BOOL* <i>pbValueArray</i>) | Query for Position Change | 45 |
| BOOL PI_HDT (int <i>ID</i> , const int* <i>iDeviceIDsArray</i> , const int* <i>iAxisIDsArray</i> , const int* <i>piValueArray</i> , int <i>iArraySize</i>) | Set HID Default Lookup Table | 46 |
| BOOL PI_HIA (int <i>ID</i> , const char* <i>szAxes</i> , const int* <i>iFunctionArray</i> , const int* <i>iDeviceIDsArray</i> , const int* <i>iAxesIDsArray</i>) | Configure Control Done By HID Axis | 46 |
| BOOL PI_HIL (int <i>ID</i> , const int* <i>iDeviceIDsArray</i> , const int* <i>iLED_IDsArray</i> , const int* <i>pnValueArray</i> , int <i>iArraySize</i>) | Set State Of HID LED | 46 |
| BOOL PI_HIN (int <i>ID</i> , const char* <i>szAxes</i> , const BOOL* <i>pbValueArray</i>) | Set Activation State For HID Control | 47 |
| BOOL PI_HIS (int <i>ID</i> , const int* <i>iDeviceIDsArray</i> , const int* <i>iItemIDsArray</i> , const int* <i>iPropertyIDArray</i> , const char* <i>szValues</i> , int <i>iArraySize</i>) | Configure HI Device | 47 |
| BOOL PI_HIT (int <i>ID</i> , const int* <i>piTableIDsArray</i> , const int* <i>piPointNumberArray</i> , const double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Fill HID Lookup Table | 47 |
| BOOL PI_HLT (int <i>ID</i> , const char* <i>szAxes</i>) | Halt Motion Smoothly | 47 |
| BOOL PI_IFC (int <i>ID</i> , const char* <i>szParameters</i> , const char* <i>szValues</i>) | Interface configuration in volatile memory | 48 |
| BOOL PI_IFS (int <i>ID</i> , const char* <i>szPassword</i> , const char* <i>szParameters</i> , const char* <i>szValues</i>) | Interface parameter store in non-volatile memory | 48 |
| BOOL PI_IMP (int <i>ID</i> , const char* <i>szAxes</i> , const double* <i>pdImpulseSize</i>) | Starts performing an impulse and recording the impulse response | 49 |
| BOOL PI_IsControllerReady (int <i>ID</i> , int* <i>piControllerReady</i>) | Asks controller for ready status | 49 |

| Function | Short Description | Page |
|--|--|------|
| BOOL PI_IsMoving (int <i>ID</i> , const char* <i>szAxes</i> , BOOL* <i>pbValueArray</i>) | Check if axes are moving | 49 |
| BOOL PI_IsRunningMacro (int <i>ID</i> , BOOL* <i>pbRunningMacro</i>) | Check if controller is currently running a macro | 49 |
| BOOL PI_JAX (int <i>ID</i> , int <i>iJoystickID</i> , const int <i>iAxesID</i> , const char* <i>szAxesIDs</i>) | Set Axis Controlled By Joystick | 49 |
| BOOL PI_JDT (int <i>ID</i> , const int* <i>iJoystickIDsArray</i> , const int* <i>iAxisIDsArray</i> , const int* <i>piValArray</i> , int <i>iArraySize</i>) | Set Joystick Default Lookup Table | 50 |
| BOOL PI_JLT (int <i>ID</i> , int <i>iJoystickID</i> , int <i>iAxisID</i> , int <i>iStartAddress</i> , const double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Fill Joystick Lookup Table | 50 |
| BOOL PI_JON (int <i>ID</i> , const int* <i>iJoystickIDsArray</i> , const BOOL* <i>pbValArray</i> , int <i>iArraySize</i>) | Set Joystick Activation Status | 50 |
| BOOL PI_KCP (int <i>ID</i> , const char* <i>szSource</i> , const char* <i>szDestination</i>) | Copies a coordinate system | 51 |
| BOOL PI_KEN (int <i>ID</i> , const char* <i>szNameOfCoordSystem</i>) | Enables an already defined coordinate system | 51 |
| BOOL PI_KLD (int <i>ID</i> , const char* <i>szNameOfCoordSystem</i> , const char* <i>szAxes</i> , const double* <i>pdValueArray</i>) | Defines a levelling coordinate system (KLD type) | 51 |
| BOOL PI_KLF (int <i>ID</i> , const char* <i>szNameOfCoordSystem</i>) | Defines a levelling coordinate system (KLF type) | 51 |
| BOOL PI_KLN (int <i>ID</i> , const char* <i>szNameOfChild</i> , const char* <i>szNameOfParent</i>) | Links two coordinate systems | 51 |
| BOOL PI_KRM (int <i>ID</i> , const char* <i>szNameOfCoordSystem</i>) | Deletes a coordinate system | 52 |
| BOOL PI_KSB (int <i>ID</i> , const char* <i>szNameOfCoordSystem</i> , const char* <i>szAxes</i> , const double* <i>pdValueArray</i>) | Defines a coordinate system of KSB type | 52 |
| BOOL PI_KSD (int <i>ID</i> , const char* <i>szNameOfCoordSystem</i> , const char* <i>szAxes</i> , const double* <i>pdValueArray</i>) | Defines a coordinate system of KSD type | 52 |
| BOOL PI_KSF (int <i>ID</i> , const char* <i>szNameOfCoordSystem</i>) | Defines a coordinate system of KSF type | 52 |
| BOOL PI_KST (int <i>ID</i> , const char* <i>szNameOfCoordSystem</i> , const char* <i>szAxes</i> , const double* <i>pdValueArray</i>) | Defines a coordinate system of KST type | 53 |
| BOOL PI_KSW (int <i>ID</i> , const char* <i>szNameOfCoordSystem</i> , const char* <i>szAxes</i> , const double* <i>pdValueArray</i>) | Defines a coordinate system of KSW type | 53 |
| BOOL PI_MAC_BEG (int <i>ID</i> , const char* <i>szMacroName</i>) | Call Macro Function: Start recording macro | 53 |
| BOOL PI_MAC_DEF (int <i>ID</i> , const char* <i>szMacroName</i>) | Call Macro Function: Set the specified macros as start-up macro | 53 |
| BOOL PI_MAC_DEL (int <i>ID</i> , const char* <i>szMacroName</i>) | Call Macro Function: Delete macro | 54 |
| BOOL PI_MAC_END (int <i>ID</i>) | Call Macro Function: End macro recording | 54 |
| BOOL PI_MAC_NSTART (int <i>ID</i> , const char* <i>szMacroName</i> , int <i>nrRuns</i>) | Call Macro Function: Execute macro n times | 54 |
| BOOL PI_MAC_NSTART_Args (int <i>ID</i> , const char* <i>szMacroName</i> , int <i>nrRuns</i> , const char* <i>szArgs</i>) | Call Macro Function: Execute macro n times using variable | 54 |
| BOOL PI_MAC_qDEF (int <i>ID</i> , char* <i>szBuffer</i> , int <i>iBufferSize</i>) | Call Macro Function: Ask name of start-up macro | 55 |

| Function | Short Description | Page |
|--|---|------|
| BOOL PI_MAC_qERR (int <i>ID</i> , char* <i>szBuffer</i> , int <i>iBufferSize</i>) | Call Macro Function: Get Error Number | 55 |
| BOOL PI_MAC_qFREE (int <i>ID</i> , long* <i>pnFreeSpace</i>) | Call Macro Function: Get Free Memory Space | 55 |
| BOOL PI_MAC_START (int <i>ID</i> , const char * <i>szMacroName</i>) | Call Macro Function: Start macro (single run) | 55 |
| BOOL PI_MAC_START_Args (int <i>ID</i> , const char* <i>szMacroName</i> , const char* <i>szArgs</i>) | Call Macro Function: Start macro (single run) using variable | 55 |
| BOOL PI_MEX (int <i>ID</i> , const char * <i>szCondition</i>) | Stop Macro Execution Due To Condition | 56 |
| BOOL PI_MOD (int <i>ID</i> , const char* <i>szItems</i> , const unsigned int* <i>iModeArray</i> , const char* <i>szValues</i>); | Set Mode | 56 |
| BOOL PI_MOV (int <i>ID</i> , const char* <i>szAxes</i> , const double* <i>pdValueArray</i>) | Set Target Position | 56 |
| BOOL PI_MRT (int <i>ID</i> , const char* <i>szAxes</i> , const double* <i>pdValueArray</i>) | Set target relative to current position and orientation in Tool coordinate system | 56 |
| BOOL PI_MRW (int <i>ID</i> , const char* <i>szAxes</i> , const double* <i>pdValueArray</i>) | Set target relative to current position and orientation in Work coordinate system | 57 |
| BOOL PI_MVE (int <i>ID</i> , const char* <i>szAxes</i> , const double* <i>pdValueArray</i>) | Set Target Position for Vector Move | 57 |
| BOOL PI_MVR (int <i>ID</i> , const char* <i>szAxes</i> , const double* <i>pdValueArray</i>) | Set Target Relative To Current Position | 57 |
| BOOL PI_NAV (int <i>ID</i> , const int* <i>piAnalogChannelIds</i> , const int* <i>piNrReadingsValues</i> , int <i>iArraySize</i>) | Set Number of Readout Values to be Averaged | 58 |
| BOOL PI_NLM (int <i>ID</i> , const char* <i>szAxes</i> , const double* <i>pdValueArray</i>) | Set Low Position Soft Limit | 58 |
| BOOL PI_OAC (int <i>ID</i> , const int* <i>piPIEZOWALKChannelsArray</i> , const double* <i>pdValueArray</i> , int <i>iArraySize</i>); | Set open-loop acceleration | 58 |
| BOOL PI_OAD (int <i>ID</i> , const int* <i>piPIEZOWALKChannelsArray</i> , const double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Open-loop analog driving | 58 |
| BOOL PI_ODC (int <i>ID</i> , const int* <i>piPIEZOWALKChannelsArray</i> , const double* <i>pdValueArray</i> , int <i>iArraySize</i>); | Set open-loop deceleration | 59 |
| BOOL PI_OMA (int <i>ID</i> , const char* <i>szAxes</i> , const double* <i>pdValueArray</i>); | Absolute open-loop motion | 59 |
| BOOL PI_OMR (int <i>ID</i> , const char* <i>szAxes</i> , const double* <i>pdValueArray</i>); | Relative open-loop motion | 59 |
| BOOL PI_ONL (int <i>ID</i> , const int* <i>iPiezoCannels</i> , const int* <i>pdValarray</i> , int <i>iArraySize</i>) | Sets control mode for piezo channel | 59 |
| BOOL PI_OSM (int <i>ID</i> , const int* <i>piPIEZOWALKChannelsArray</i> , const int* <i>piValueArray</i> , int <i>iArraySize</i>) | Open-loop step moving (using full step cycles) | 60 |
| BOOL PI_OSMf (int <i>ID</i> , const int* <i>piPIEZOWALKChannelsArray</i> , const double* <i>pdValueArray</i> , int <i>iArraySize</i>); | Open-loop step moving (allowing also parts of a step cycle) | 60 |
| BOOL PI_OVL (int <i>ID</i> , const int* <i>piPIEZOWALKChannelsArray</i> , double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Set velocity for open-loop nanostepping motion | 60 |
| BOOL PI_PLM (int <i>ID</i> , const char* <i>szAxes</i> , const double* <i>pdValueArray</i>) | Set High Position Soft Limit | 61 |
| BOOL PI_POS (int <i>ID</i> , const char* <i>szAxes</i> , const double* <i>pdValueArray</i>) | Set Real Position | 61 |
| BOOL PI_qACC (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Get Closed-Loop Acceleration | 61 |

| Function | Short Description | Page |
|---|--|------|
| BOOL PI_qAOS (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Get Analog Input Offset | 61 |
| BOOL PI_qATC (int <i>ID</i> , const int* <i>piChannels</i> , int* <i>piValueArray</i> , int <i>iArraySize</i>) | Get the options used for auto calibration procedure | 62 |
| BOOL PI_qATZ (int <i>ID</i> , const char* <i>szAxes</i> , int* <i>piAtzResult</i>) | Reports if AutoZero procedure was successful | 62 |
| BOOL PI_qATS (int <i>ID</i> , const int* <i>piChannels</i> , const int* <i>piOptions</i> , int* <i>piValueArray</i> , int <i>iArraySize</i>) | Query results of the latest auto calibration procedure | 62 |
| BOOL PI_qBRA (const int <i>ID</i> , char * <i>szBuffer</i> , const int <i>maxlen</i>) | Query brake state (on/off) | 62 |
| BOOL PI_qCAV (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Get current value of controlled variable | 63 |
| BOOL PI_qCCL (int <i>ID</i> , int* <i>piComandLevel</i>) | Get current command level | 63 |
| BOOL PI_qCCV (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Get currently valid control value | 63 |
| BOOL PI_qCMN (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Get the minimum commandable closed-loop target | 63 |
| BOOL PI_qCMO (int <i>ID</i> , const char* <i>szAxes</i> , int* <i>piValueArray</i>) | Get closed-loop control mode | 63 |
| BOOL PI_qCMX (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Get the maximum commandable closed-loop target | 64 |
| BOOL PI_qCOV (int <i>ID</i> , const int* <i>piChannelsArray</i> , double* <i>pdValueArray</i> , int <i>iArraySize</i>); | Get Open-Loop Velocity | 64 |
| BOOL PI_qCST (int <i>ID</i> , const char* <i>szAxes</i> , char* <i>szNames</i> , int <i>iBufferSize</i>) | Get Stage Type Of Selected Axis | 64 |
| BOOL PI_qCSV (int <i>ID</i> , double* <i>pdCommandSyntaxVersion</i>) | Get Current Syntax Version | 64 |
| BOOL PI_qCTI (int <i>ID</i> , const int* <i>piTriggerInputIds</i> , const int* <i>piTriggerParameterArray</i> , char* <i>szValueArray</i> , int <i>iArraySize</i> , int <i>iBufferSize</i>) | Get Trigger Input configuration | 65 |
| BOOL PI_qCTO (int <i>ID</i> , const int* <i>piTriggerOutputIdsArray</i> , const int* <i>piTriggerParameterArray</i> , double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Get Trigger Output configuration | 65 |
| BOOL PI_qCTOString (int <i>ID</i> , const int* <i>piTriggerOutputIds</i> , const int* <i>piTriggerParameterArray</i> , char* <i>szValueArray</i> , int <i>iArraySize</i> , int <i>iBufferSize</i>) | Get Trigger Output configuration | 65 |
| BOOL PI_qCTV (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Get Target Values | 65 |
| BOOL PI_qDCO (int <i>ID</i> , const char* <i>szAxes</i> , BOOL* <i>pbValueArray</i>) | Gets drift compensation mode | 66 |
| BOOL PI_qDEC (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Get Closed-Loop Deceleration | 66 |
| BOOL PI_qDFH (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Returns current home position definition | 66 |
| BOOL PI_qDIO (int <i>ID</i> , const long* <i>piChannelsArray</i> , BOOL* <i>pbValueArray</i> , int <i>iArraySize</i>) | Get Digital Input Lines | 66 |
| BOOL PI_qDRC (int <i>ID</i> , const int* <i>piRecordTableIdsArray</i> , char* <i>szRecordSourceIds</i> , int* <i>piRecordOptionArray</i> , int <i>iRecordSourceIdsBufferSize</i> , int <i>iRecordOptionArraySize</i>) | Get Data Recorder Configuration | 67 |
| BOOL PI_qDRL (int <i>ID</i> , const int* <i>piRecordTableIdsArray</i> , int* <i>piNumberOfRecordedValuesArray</i> , int <i>iArraySize</i>) | Reads the number of points comprised by the last recording | 67 |

| Function | Short Description | Page |
|---|--|------|
| BOOL PI_qDRR (int <i>ID</i> , const int* <i>piRecTableIdsArray</i> , int <i>iNumberOfRecTables</i> , int <i>iOffsetOfFirstPointInRecordTable</i> , int <i>iNumberOfValues</i> , double** <i>pdValueArray</i> , char* <i>szGcsArrayHeader</i> , int <i>iGcsArrayHeaderMaxSize</i>) | Get Recorded Data Values | 67 |
| BOOL PI_qDRR_SYNC (int <i>ID</i> , int <i>iRecordTableId</i> , int <i>iOffsetOfFirstPointInRecordTable</i> , int <i>iNumberOfValues</i> , double* <i>pdValueArray</i>) | Get Recorded Data Values | 68 |
| BOOL PI_qDRT (int <i>ID</i> , const int* <i>piRecordTableIdsArray</i> , int* <i>piTriggerSourceArray</i> , char* <i>szValues</i> , int <i>iArraySize</i> , int <i>iValueBufferLength</i>) | Get Data Recorder Trigger Source | 68 |
| BOOL PI_qEAX (int <i>ID</i> , const char* <i>szAxes</i> , BOOL* <i>pbValueArray</i>) | Get Enable Status Of Axes | 68 |
| BOOL PI_qECO (int <i>ID</i> , const char* <i>szSendString</i> , char* <i>szValues</i> , int <i>iBufferSize</i>) | Echo a String | 69 |
| BOOL PI_qERR (int <i>ID</i> , long* <i>pnError</i>) | Get Error Number | 69 |
| BOOL PI_qFGC (int <i>ID</i> , const char* <i>szProcessIds</i> , double* <i>pdScanAxisCenterValueArray</i> , double* <i>pdStepAxisCenterValueArray</i>) | Fast alignment: Gets the current center position of the circular motion of a gradient search routine | 69 |
| BOOL PI_qFPH (int <i>ID</i> , const char* <i>szAxes</i> , BOOL* <i>pbValueArray</i>) | Get Phase Offset | 69 |
| BOOL PI_qFRC (int <i>ID</i> , const char* <i>szProcessIdsBase</i> , char* <i>szBuffer</i> , int <i>iBufferSize</i>) | Fast alignment: Gets coupled fast alignment routines | 70 |
| BOOL PI_qFRF (int <i>ID</i> , const char* <i>szAxes</i> , BOOL* <i>pbValueArray</i>) | Get Referencing Result | 70 |
| BOOL PI_qFRH (int <i>ID</i> , char* <i>szBuffer</i> , int <i>iBufferSize</i>) | Fast alignment: Lists help string for routine results | 70 |
| BOOL PI_qFRP (int <i>ID</i> , const char* <i>szScanRoutineNames</i> , int* <i>piOptionsArray</i>) | Fast alignment: Gets the current state of a fast alignment routine | 70 |
| BOOL PI_qFRR (int <i>ID</i> , const char* <i>szScanRoutineNames</i> , const unsigned int* <i>iResultIdsArray</i> , char* <i>szResult</i> , int <i>iBufferSize</i>) | Fast alignment: Gets the results of a fast alignment routine | 71 |
| BOOL PI_qFSS (int <i>ID</i> , int* <i>piResult</i>) | Get Status of Fast Scan Routines | 71 |
| BOOL PI_qHAR (int <i>ID</i> , const char* <i>szAxes</i> , BOOL* <i>pbValueArray</i>) | Indicate Hard Stops | 71 |
| BOOL PI_qHDR (int <i>ID</i> , char* <i>szBuffer</i> , int <i>iBufferSize</i>) | Get All Data Recorder Options | 71 |
| BOOL PI_qHDT (int <i>ID</i> , const int* <i>iDeviceIdsArray</i> , const int* <i>iAxisIdsArray</i> , int* <i>piValueArray</i> , int <i>iArraySize</i>) | Get HID Default Lookup Table | 72 |
| BOOL PI_qHIA (int <i>ID</i> , const char* <i>szAxes</i> , const int* <i>iFunctionArray</i> , int* <i>iDeviceIdsArray</i> , int* <i>iAxisIdsArray</i>) | Get Configuration Of Control Done By HID Axis | 72 |
| BOOL PI_qHIB (int <i>ID</i> , const int* <i>iDeviceIdsArray</i> , const int* <i>iButtonIdsArray</i> , int* <i>pbValueArray</i> , int <i>iArraySize</i>) | Get State Of HID Button | 72 |
| BOOL PI_qHIE (int <i>ID</i> , const int* <i>iDeviceIdsArray</i> , const int* <i>iAxisIdsArray</i> , double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Get Deflection Of HID Axis | 73 |
| BOOL PI_qHIL (int <i>ID</i> , const int* <i>iDeviceIdsArray</i> , const int* <i>iLED_IdsArray</i> , int* <i>pnValueArray</i> , int <i>iArraySize</i>) | Get State Of HID LED | 73 |
| BOOL PI_qHIN (int <i>ID</i> , const char* <i>szAxes</i> , BOOL* <i>pbValueArray</i>) | Get Activation State Of HID Control | 73 |

| Function | Short Description | Page |
|---|---|------|
| BOOL PI_qHIS (int <i>ID</i> , char* <i>szBuffer</i> , int <i>iBufferSize</i>) | Get Configuration Of HI Device | 73 |
| BOOL PI_qHIT (int <i>ID</i> , const int* <i>piTableIdsArray</i> , int <i>iNumberOfTables</i> , int <i>iOffsetOfFirstPointInTable</i> , int <i>iNumberOfValues</i> , double** <i>pdValueArray</i> , char* <i>szGcsArrayHeader</i> , int <i>iGcsArrayHeaderMaxSize</i>) | Get HID Lookup Table Values | 74 |
| BOOL PI_qHLP (int <i>ID</i> , char* <i>szBuffer</i> , int <i>iBufferSize</i>) | Get List of Available Commands | 74 |
| BOOL PI_qHPA (int <i>ID</i> , char* <i>szBuffer</i> , int <i>iBufferSize</i>) | Get List of Available Parameters | 74 |
| BOOL PI_qHPV (int <i>ID</i> , char* <i>szBuffer</i> , int <i>iBufferSize</i>) | Get Help String with Possible Parameters Values | 74 |
| BOOL PI_qIDN (int <i>ID</i> , char* <i>szBuffer</i> , int <i>iBufferSize</i>) | Get Device Identification | 75 |
| BOOL PI_qIFC (int <i>ID</i> , const char* <i>szParameters</i> , char* <i>szBuffer</i> , int <i>iBufferSize</i>) | Interface configuration in volatile memory | 75 |
| BOOL PI_qIFS (int <i>ID</i> , const char* <i>szParameters</i> , char* <i>szBuffer</i> , int <i>iBufferSize</i>) | Interface parameter store in non-volatile memory | 75 |
| BOOL PI_qIMP (nt <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Starts performing an impulse and recording the impulse response | 75 |
| BOOL PI_qJAS (int <i>ID</i> , const int* <i>iJoystickIDsArray</i> , const int <i>iAxesIDsArray</i> , double* <i>pdValarray</i> , int <i>iArraySize</i>) | Query Joystick Axis Status | 76 |
| BOOL PI_qJAX (int <i>ID</i> , const int* <i>iJoystickIDsArray</i> , const int* <i>iAxesIDsArray</i> , int <i>iArraySize</i> , char* <i>szAxesBuffer</i> , int <i>iBufferSize</i>) | Get Axis Controlled By Joystick | 76 |
| BOOL PI_qJBS (int <i>ID</i> , const int* <i>iJoystickIDsArray</i> , const int* <i>iButtonIDsArray</i> , BOOL* <i>pbValarray</i> , int <i>iArraySize</i>) | Query Joystick Button Status | 76 |
| BOOL PI_qJLT (int <i>ID</i> , const int* <i>iJoystickIDsArray</i> , const int* <i>iAxisIDsArray</i> , int <i>iNumberOfTables</i> , int <i>iOffsetOfFirstPointInTable</i> , int <i>iNumberOfValues</i> , double** <i>pdValueArray</i> , char* <i>szGcsArrayHeader</i> , int <i>iGcsArrayHeaderMaxSize</i>) | Get Joystick Lookup Table Values | 77 |
| BOOL PI_qJON (int <i>ID</i> , const int* <i>iJoystickIDsArray</i> , BOOL* <i>pbValarray</i> , int <i>iArraySize</i>) | Get Joystick Activation Status | 77 |
| BOOL PI_qKEN (int <i>ID</i> , const char* <i>szNamesOfCoordSystems</i> , char* <i>buffer</i> , int <i>bufsize</i>) | List enabled coordinate systems by name | 77 |
| BOOL PI_qKET (int <i>ID</i> , const char* <i>szTypes</i> , char* <i>buffer</i> , int <i>bufsize</i>) | List enabled coordinate systems by type | 78 |
| BOOL PI_qKLC (int <i>ID</i> , const char* <i>szNameOfCoordSystem1</i> , const char* <i>szNameOfCoordSystem2</i> , const char* <i>szItem1</i> , const char* <i>szItem2</i> , char* <i>buffer</i> , int <i>bufsize</i>) | List properties of available Work/Tool combinations | 78 |
| BOOL PI_qKLN (int <i>ID</i> , const char* <i>szNamesOfCoordSystems</i> , char* <i>buffer</i> , int <i>bufsize</i>) | List coordinate system chains | 78 |
| BOOL PI_qKLS (int <i>ID</i> , const char* <i>szNameOfCoordSystem</i> , const char* <i>szItem1</i> , const char* <i>szItem2</i> , char* <i>buffer</i> , int <i>bufsize</i>) | List properties of all defined coordinate systems | 78 |
| BOOL PI_qKLT (int <i>ID</i> , const char* <i>szStartCoordSystem</i> , const char* <i>szEndCoordSystem</i> , char* <i>buffer</i> , int <i>bufsize</i>) | Get the resulting coordinate system of a chain | 79 |
| BOOL PI_qLIM (int <i>ID</i> , const char* <i>szAxes</i> , BOOL* <i>pbValueArray</i>) | Indicate Limit Switches | 79 |
| BOOL PI_qMAC (int <i>ID</i> , const char* <i>szMacroName</i> , char* <i>szBuffer</i> , int <i>iBufferSize</i>) | List Macros | 79 |

| Function | Short Description | Page |
|--|---|------|
| BOOL PI_qMAN (int <i>ID</i> , const char* <i>szCommand</i> , char* <i>szBuffer</i> , int <i>iBufferSize</i>) | Get Help String For Command | 80 |
| BOOL PI_qMOD (int <i>ID</i> , const char* <i>szItems</i> , const unsigned int* <i>iModeArray</i> , char* <i>szValues</i> , int <i>iMaxValuesSize</i>) | Get Mode | 80 |
| BOOL PI_qMOV (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Get Target Position | 80 |
| BOOL PI_qNAV (int <i>ID</i> , const int* <i>piAnalogChannelIds</i> , int* <i>piNrReadingsValues</i> , int <i>iArraySize</i>) | Get Number of Readings to be Averaged | 80 |
| BOOL PI_qNLM (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Get Low Position Soft Limit | 81 |
| BOOL PI_qOAC (int <i>ID</i> , const int* <i>piPIEZOWALKChannelsArray</i> , double* <i>pdValueArray</i> , int <i>iArraySize</i>); | Get open-loop acceleration | 81 |
| BOOL PI_qOAD (int <i>ID</i> , const int* <i>piPIEZOWALKChannelsArray</i> , double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Get Voltage For Open-Loop Analog Motion | 81 |
| BOOL PI_qODC (int <i>ID</i> , const int* <i>piPIEZOWALKChannelsArray</i> , double* <i>pdValueArray</i> , int <i>iArraySize</i>); | Get Open-Loop Deceleration | 81 |
| BOOL PI_qOMA (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>); | Get Open-Loop Target Position | 82 |
| BOOL PI_qONL (int <i>ID</i> , const int* <i>iPiezoCannels</i> , int* <i>pdValarray</i> , int <i>iArraySize</i>) | Get Control Mode | 82 |
| BOOL PI_qONT (int <i>ID</i> , const char* <i>szAxes</i> , BOOL* <i>pbValueArray</i>) | Get On Target State | 82 |
| BOOL PI_qOSN (int <i>ID</i> , const int* <i>piPiezoWalkChannelsArray</i> , double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Get number of steps still to be performed | 82 |
| BOOL PI_qOVF (int <i>ID</i> , const char* <i>szAxes</i> , BOOL* <i>pbOverflow</i>) | Checks overflow status | 82 |
| BOOL PI_qOVL (int <i>ID</i> , const int* <i>piPIEZOWALKChannelsArray</i> , double* <i>pdValueArray</i>) | Get Open-Loop Velocity | 83 |
| BOOL PI_qPLM (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Get High Position Soft Limit | 83 |
| BOOL PI_qPOS (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Get Real Position | 83 |
| BOOL PI_qPUN (int <i>ID</i> , const char* <i>szAxes</i> , char* <i>szUnit</i> , int <i>iBufferSize</i>) | Get the Position Units | 83 |
| BOOL PI_qRMC (int <i>ID</i> , char * <i>szBuffer</i> , int <i>iBufferSize</i>) | List Running Macros | 84 |
| BOOL PI_qRON (int <i>ID</i> , const char* <i>szAxes</i> , BOOL* <i>pbValueArray</i>) | Get Reference Mode | 84 |
| BOOL PI_qRTO (int <i>ID</i> , const char* <i>szAxes</i> , int* <i>pbValueArray</i>) | Read the "ready-for-turn-off state" | 84 |
| BOOL PI_qRTR (int <i>ID</i> , int* <i>piRecordTableRate</i>) | Get Record Table Rate | 84 |
| BOOL PI_qSAI (int <i>ID</i> , char* <i>szAxes</i> , int <i>iBufferSize</i>) | Get List Of Current Axis Identifiers | 85 |
| BOOL PI_qSAI_ALL (int <i>ID</i> , char* <i>szAxes</i> , int <i>iBufferSize</i>) | Get List Of Current Axis Identifiers | 85 |
| BOOL PI_qSCT (int <i>ID</i> , double* <i>pdCycleTime</i>) | Get Cycle Time | 85 |
| BOOL PI_qSEP (int <i>ID</i> , const char* <i>szAxes</i> , const int* <i>piParameterArray</i> , double* <i>pdValueArray</i> , char* <i>szStrings</i> , int <i>iMaximumStringSize</i>) | Get Nonvolatile Memory Parameters | 85 |

| Function | Short Description | Page |
|---|--|------|
| BOOL PI_qSGA (int <i>ID</i> , const int* <i>piAnalogChannelIds</i> , int* <i>piGainValues</i> , int <i>iArraySize</i>) | Get Gain | 86 |
| BOOL PI_qSIC (int <i>ID</i> , const int* <i>piFastAlignmentInputIdsArray</i> , int <i>iNumberOfInputIds</i> , char* <i>szBuffer</i> , int <i>iBufferSize</i>) | Fast alignment: Gets the calculation settings for the given fast alignment input channel | 86 |
| BOOL PI_qSMO (int <i>ID</i> , char *const <i>szAxes</i> , int * <i>pnValueArray</i>) | Get Control Value | 86 |
| BOOL PI_qSPA (int <i>ID</i> , const char* <i>szAxes</i> , unsigned int* <i>piParameterArray</i> , double* <i>pdValueArray</i> , char* <i>szStrings</i> , int <i>iMaxNameSize</i>) | Get Volatile Memory Parameters | 86 |
| BOOL PI_qSPI (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Get Pivot Point | 87 |
| BOOL PI_qSRG (int <i>ID</i> , const char* <i>szAxes</i> , const int* <i>iRegisterArray</i> , int * <i>iValArray</i>) | Query Status Register Value | 87 |
| BOOL PI_qSSA (int <i>ID</i> , const int* <i>iPIEZOWALKChannels</i> , double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Get Step Amplitude | 87 |
| BOOL PI_qSSL (int <i>ID</i> , const char* <i>szAxes</i> , BOOL* <i>pbValueArray</i>) | Get Soft Limit Status | 87 |
| BOOL PI_qSSN (int <i>ID</i> , char* <i>szSerialNumber</i> , int <i>iBufferSize</i>) | Get Device Serial Number | 88 |
| BOOL PI_qSST (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Get Step Size | 88 |
| BOOL PI_qSTE (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Get last sent step size | 88 |
| BOOL PI_qSVA (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Returns last valid open-loop control value | 88 |
| BOOL PI_qSVO (int <i>ID</i> , const char* <i>szAxes</i> , BOOL* <i>pbValueArray</i>) | Get Servo State (Open-Loop / Closed-Loop Operation) | 89 |
| BOOL PI_qTAC (int <i>ID</i> , int * <i>piNrChannels</i>) | Tell Analog Channels | 89 |
| BOOL PI_qTAD (int <i>ID</i> , const int* <i>piSensorChannelsArray</i> , int* <i>piValueArray</i> , int <i>iArraySize</i>) | Returns ADC value | 89 |
| BOOL PI_qTAV (int <i>ID</i> , const int* <i>piChannelsArray</i> , double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Get Analog Input Voltage | 89 |
| BOOL PI_qTCI (int <i>ID</i> , const int* <i>piFastAlignmentInputIdsArray</i> , double* <i>pdCalculatedInputValueArray</i> , int <i>iArraySize</i>) | Fast alignment: Gets calculated value of given fast alignment input channel. | 89 |
| BOOL PI_qTCV (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Get Commanded Closed-Loop Velocity | 90 |
| BOOL PI_qTIO (int <i>ID</i> , int* <i>piInputNr</i> , int* <i>piOutputNr</i>) | Tell Digital I/O Lines | 90 |
| BOOL PI_qTMN (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Get Minimum Commandable Position | 90 |
| BOOL PI_qTMX (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Get Maximum Commandable Position | 90 |
| BOOL PI_qTNR (int <i>ID</i> , int* <i>piNumberOfRecordTables</i>) | Get Number Of Record Tables | 91 |
| BOOL PI_qTNS (int <i>ID</i> , const int* <i>piSensorChannelsArray</i> , double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Returns normalized sensor value for the specified sensor number | 91 |
| BOOL PI_qTPC (int <i>ID</i> , int* <i>piNumberOfPiezoChannels</i>) | Get the number of output signal channels available on the controller. | 91 |
| BOOL PI_qTRA (int <i>ID</i> , const char* <i>szAxes</i> , const double* <i>pdComponents</i> , double* <i>pdValueArray</i>) | Get maximum absolute position which can be reached from the current position | 91 |

| Function | Short Description | Page |
|---|--|------|
| BOOL PI_qTRI (int <i>ID</i> , const int* <i>piTriggerInputIds</i> , BOOL* <i>pbTriggerState</i> , int <i>iArraySize</i>) | Get Trigger Input State | 92 |
| BOOL PI_qTRO (int <i>ID</i> , const int* <i>piTriggerOutputIds</i> , BOOL* <i>pbTriggerState</i> , int <i>iArraySize</i>) | Get Trigger Output State | 92 |
| BOOL PI_qTRS (nt <i>ID</i> , const char* <i>szAxes</i> , BOOL * <i>pbValueArray</i>) | Indicate Reference Switch | 92 |
| BOOL PI_qTSC (int <i>ID</i> , int* <i>piNumberOfSensorChannels</i>) | Get the number of input signal channels available on the controller. | 92 |
| BOOL PI_qTSP (int <i>ID</i> , const int* <i>piSensorChannelsArray</i> , double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Requests the current position of the given input signal channel | 93 |
| BOOL PI_qTVI (int <i>ID</i> , char * <i>szBuffer</i> , int <i>iBufferSize</i>) | Tell Valid Character Set For Axis Identifiers | 93 |
| BOOL PI_qVAR (int <i>ID</i> , const char* <i>szVariables</i> , char* <i>szValues</i> ,int <i>iBufferSize</i>) | Get Variable Values | 93 |
| BOOL PI_qVCO (int <i>ID</i> , char* <i>szAxes</i> , BOOL* <i>pbValueArray</i>) | Get Velocity Control Mode | 93 |
| BOOL PI_qVEL (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Get Closed-Loop Velocity | 94 |
| BOOL PI_qVER (int <i>ID</i> , char* <i>szVersion</i> , int <i>iBufferSize</i>) | Get Version | 94 |
| BOOL PI_qVLS (int <i>ID</i> , double* <i>pdSystemVelocity</i>) | Get System Velocity | 94 |
| BOOL PI_qVMA (int <i>ID</i> , const int* <i>piPiezoChannelsArray</i> , double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Get Voltage Output High Limit | 94 |
| BOOL PI_qVMI (int <i>ID</i> , const int* <i>piPiezoChannelsArray</i> , double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Get Voltage Output Low Limit | 94 |
| BOOL PI_qVMO (int <i>ID</i> , const char* <i>szAxes</i> , const double* <i>pdValarray</i> , BOOL* <i>pbMovePossible</i>) | Virtual Move | 95 |
| BOOL PI_qVOL (int <i>ID</i> , const int* <i>piPiezoChannelsArray</i> , double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Get current piezo output voltages | 95 |
| BOOL PI_qVST (int <i>ID</i> , char * <i>szBuffer</i> , int <i>iBufferSize</i>) | Get the names of the available stage types | 95 |
| BOOL PI_RBT (int <i>ID</i>) | Reboot System | 96 |
| BOOL PI_RNP (int <i>ID</i> , const int* <i>piPIEZOWALKChannelsArray</i> , const double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Relax PiezoWalk Piezos | 96 |
| BOOL PI_RON (const int <i>ID</i> , char *const <i>szAxes</i> , BOOL * <i>pbValarray</i>) | Set Reference Mode | 96 |
| BOOL PI_RPA (int <i>ID</i> , const char* <i>szAxes</i> , const int* <i>piParameterArray</i>) | Reset Volatile Memory Parameters | 96 |
| BOOL PI_RTO (int <i>ID</i> , const char* <i>szAxes</i>) | Make Ready to Turn OFF | 97 |
| BOOL PI_RTR (int <i>ID</i> , int <i>iRecordTableRate</i>) | Set Record Table Rate | 97 |
| BOOL PI_SAI (int <i>ID</i> , char *const <i>szOldAxes</i> , char *const <i>szNewAxes</i>) | Set Current Axis Identifiers | 97 |
| BOOL PI_SCT (int <i>ID</i> , double <i>dCycleTime</i>) | Set Cycle Time | 97 |
| BOOL PI_SEP (int <i>ID</i> , const char* <i>szPassword</i> , const char* <i>szAxes</i> , const int* <i>piParameterArray</i> , const double* <i>pdValueArray</i> , const char* <i>szStrings</i>) | Set Nonvolatile Memory Parameters | 98 |
| BOOL PI_SGA (int <i>ID</i> , const int* <i>piAnalogChannelIds</i> , const int* <i>piGainValues</i> , int <i>iArraySize</i>) | Set Gain | 98 |

| Function | Short Description | Page |
|--|--|------|
| BOOL PI_SIC (int <i>ID</i> , int <i>iFastAlignmentInputId</i> , int <i>iCalcType</i> , const double* <i>pdParameters</i> , int <i>iNumberOfParameters</i>) | Fast alignment: Defines calculation settings for the given fast alignment input channel | 98 |
| BOOL PI_SMO (int <i>ID</i> , char *const <i>szAxes</i> , int * <i>pnVaueArray</i>) | Set Open-Loop Control Value | 99 |
| BOOL PI_SPA (int <i>ID</i> , const char* <i>szAxes</i> , const unsigned int* <i>piParameterArray</i> , const double* <i>pdValueArray</i> , const char* <i>szStrings</i>) | Set Volatile Memory Parameters | 99 |
| BOOL PI_SPI (int <i>ID</i> , const char* <i>szAxes</i> , const double* <i>pdValueArray</i>) | Set Pivot Point | 99 |
| BOOL PI_SSA (int <i>ID</i> , const int* <i>piPIEZOWALKChannelsArray</i> , const double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Set Step Amplitude | 100 |
| BOOL PI_SSL (int <i>ID</i> , const char* <i>szAxes</i> , const BOOL* <i>pbValueArray</i>) | Set Soft Limit | 100 |
| BOOL PI_SST (int <i>ID</i> , const char* <i>szAxes</i> , const double* <i>pdValueArray</i>) | Set Step Size | 100 |
| BOOL PI_STE (int <i>ID</i> , const char* <i>szAxes</i> , const double* <i>pdStepSize</i>) | Start Step And Response – Measurement | 100 |
| BOOL PI_StopAll (int <i>ID</i>) | Stop All Axes | 101 |
| BOOL PI_STP (int <i>ID</i>) | Stop All Motion | 101 |
| BOOL PI_SVA (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Set absolute open-loop control value to move the axis | 101 |
| BOOL PI_SVO (int <i>ID</i> , const char* <i>szAxes</i> , const BOOL* <i>pbValueArray</i>) | Set Servo State (Open-Loop / Closed-Loop Operation) | 101 |
| BOOL PI_SVR (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>) | Set open-loop control value relative to the current open-loop control value to move the axis | 101 |
| BOOL PI_TRI (int <i>ID</i> , const int* <i>piTriggerInputIds</i> , const BOOL* <i>pbTriggerState</i> , int <i>iArraySize</i>) | Enables or disables the trigger input mode | 102 |
| BOOL PI_TRO (int <i>ID</i> , const int* <i>piTriggerOutputIds</i> , const BOOL* <i>pbTriggerState</i> , int <i>iArraySize</i>) | Enables or disables the trigger output mode | 102 |
| BOOL PI_VAR (int <i>ID</i> , const char* <i>szVariables</i> , const char* <i>szValues</i>) | Set Variable Value | 102 |
| BOOL PI_VCO (int <i>ID</i> , char* <i>szAxes</i> , BOOL* <i>pbValueArray</i>) | Set Velocity Control Mode | 102 |
| BOOL PI_VEL (int <i>ID</i> , const char* <i>szAxes</i> , const double* <i>pdValueArray</i>) | Set Closed-Loop Velocity | 103 |
| BOOL PI_VLS (int <i>ID</i> , double <i>dSystemVelocity</i>) | Set System Velocity | 103 |
| BOOL PI_VMA (int <i>ID</i> , const int* <i>piPiezoChannelsArray</i> , double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Set Voltage Output High Limit | 103 |
| BOOL PI_VMI (int <i>ID</i> , const int* <i>piPiezoChannelsArray</i> , double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Set Voltage Output Low Limit | 103 |
| BOOL PI_WAC (int <i>ID</i> , char * <i>szCondition</i>) | Wait For Condition For Macro Execution | 104 |
| BOOL PI_WPA (int <i>ID</i> , const char* <i>szPassword</i> , const char* <i>szAxes</i> , const int* <i>piParameterArray</i>) | Save Parameters To Nonvolatile Memory | 104 |

6.2. Function Description

The functions listed in this chapter are based on the commands of the GCS. You can use a function only if the corresponding command is supported by your controller. See the user manual of the controller for the supported commands.

For all details regarding the functionality and arguments of commands, see the command descriptions in the user manual of the controller.

See "Function Calls" (p. 7) for some general notes about the argument syntax.

BOOL PI_AAP (int *ID*, const char* *szAxis1*, double *dLength1*, const char* *szAxis2*, double *dLength2*, double *dAlignStep*, int *iNrRepeatedPositions*, int *iAnalogInput*)

Corresponding command: AAP

Starts a scanning procedure for better determination of the maximum intensity of an analog input signal.

The scanning procedure started with PI_AAP() corresponds to the "fine portion" of the scanning procedure that was started with PI_FSA().

Arguments:

ID ID of controller

szAxis1 first axis that defines scanning area

dLength1 length of scanning area along *szAxis1*

szAxis2 second axis that defines scanning area

dLength2 length of scanning area along *szAxis2*

dAlignStep starting value for the step size

iNrRepeatedPositions number of successful checks of the local maximum at the current position that is required for successfully completing

iAnalogInput is the identifier of the analog input signal whose maximum intensity is sought

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_ACC (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: ACC

Set the acceleration to use during moves of *szAxes*. The PI_ACC() setting only takes effect when the given axis is in closed-loop operation (servo on).

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray maximum accelerations for the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_ADD (int *ID*, const char* *szVariable*, double *value1*, double *value2*)

Corresponding command: ADD

Add two values and save the result to a variable.

Arguments:

ID ID of controller

szVariable name of variable to store the result

value1 first value to be added

value2 second value to be added

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_AOS (int *ID*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: AOS

Set an offset to the analog input for the given axis.

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray analog offset for the axes.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_ATC (int *ID*, const int* *piChannels*, const int* *piValueArray*, int *iArraySize*)

Corresponding command: ATC

Automatic calibration.

See "Calibration Settings" and the description of the ATC command in the User Manual of the controller for more information.

Arguments:

ID ID of controller

piChannels string with channels of the piezo control electronics

piValueArray comprises the settings to be calibrated.

iArraySize size of arrays

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_ATZ (int *ID*, const char* *szAxes*, const double* *pdLowVoltageArray*, const BOOL* *pbUseDefaultArray*)

Corresponding command: ATZ

Automatic zero-point calibration for *szAxes*. Sets the output voltage which is to be applied at the zero position of the axis and starts an appropriate calibration procedure.

CAUTION: The AutoZero procedure will move the axis, and the motion may cover the whole travel range. Make sure that it is safe for the stage to move.

See "AutoZero Procedure" and the description of the ATZ command in the User Manual of the controller for more information.

Arguments:

ID ID of controller

szAxes string with axes

pdLowVoltageArray Array with low voltages for the corresponding axes.

pbUseDefaultArray If **TRUE** the value in *pdLowVoltageArray* for the axis is ignored and the value stored in the controller (Autozero Low Voltage parameter, ID 0x07000A00) is used.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_BRA (const int *ID*, char *const *szAxes*, BOOL * *pbValarray*)

Corresponding command: BRA

Set brake state for *szAxes* to on (**TRUE**) or off (**FALSE**).

Arguments:

ID ID of controller

szAxes string with axes

pbValarray modes for the specified axes, **TRUE** for on, **FALSE** for off

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_CCL (int *ID*, int *iCommandLevel*, const char* *szPassWord*)

Corresponding command: CCL

If *Password* is correct, this function sets the *CommandLevel* of the controller and determines thus the availability of commands and the write access to the system parameters. Use PI_qHLP() to determine which commands are available in the current command level. PI_qHPA() lists the parameters including the information about which command level allows write access to them.

Arguments:

ID ID of controller

iCommandLevel can be

0 = the default setting, all commands provided for "normal" users are available, read access to all parameters

1 = provides additional commands and write access to level-1-parameters (commands and parameters from level 0 are included).

szPassWord password for CCL 1 is "ADVANCED", for CCL 0 no password is required

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_CMO (int *ID*, const char* *szAxes*, const int* *piValueArray*)

Corresponding command: CMO

Select closed-loop control mode. The selection determines the controlled variable (e.g. position or velocity or force).

The currently valid target value for the controlled variable can be queried with PI_qCTV(). An absolute target for the controlled variable can be set with PI_CTV(), a relative target can be set with PI_CTR(). The current value of the controlled variable can be queried with PI_qCAV().

ID ID of controller

szAxes string with axes

piValueArray modes for the specified axes

Returns:

TRUE if successful, **FALSE** otherwise (see p. 7)

BOOL PI_CPY (int *ID*, const char* *szVariable*, const char* *szCommand*)

Corresponding command: CPY

Copy a command response into a variable.

Arguments:

ID ID of controller

szVariable name of variable

szCommand query command, the result is stored in the variable given

Returns:

TRUE if successful, **FALSE** otherwise (see p. 7)

BOOL PI_CST (int *ID*, const char* *szAxes*, const char* *szNames*)

Corresponding command: CST

Loads the specific values for the *szNames* stage from a stage database (see also p. 120) and sends them to the controller so that the controller parameters are properly adjusted to the connected mechanics.

E-861, C-867, C-863, C-884, E-871:

The following actions are included:

- Sets the servo off
- Loads parameter values from stage database and sends them to the controllers RAM using PI_SPA()
- Checks the error

C-887:

Error checking is included.

With the C-887, the corresponding stage type is automatically assigned to all axes when the controller is switched on or rebooted.

The assignment of a stage type with PI_CST() is only permissible for axes A and B. In order to change the standard assignment for A and B in the volatile memory, PI_CST() can be used e.g. in a start-up macro.

The permissible stage types can be listed with the PI_qVST() function.

PI_CST() also switches on servo mode for axes A and B.

Arguments:

ID ID of controller

szAxes identifiers of the axes

szNames the names of the stages separated by '\n' ("line-feed"), the names must be present in one of the stage database files

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_CTI (int *ID*, const int* *piTriggerInputIds*, const int* *piTriggerParameterArray*, const char* *szValueArray*, int *iArraySize*)

Corresponding command: CTI

Configures the trigger input conditions for the given digital input line. Depending on the controller, the trigger input conditions will either become active immediately, or will become active when activated with PI_TRI().

Arguments:

ID ID of controller

piTriggerInputIds is an array with the trigger input lines of the controller

piTriggerParameterArray is an array with the CTI parameter IDs

szValueArray is a list of the values to which the CTI parameters are set. The single values must be separated by a linefeed character

iArraySize is the size of the array *piTriggerInputIds*

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_CTO (int *ID*, const int* *piTriggerOutputIds*, const int* *piTriggerParameterArray*, const double* *pdValueArray*, int *iArraySize*)

Corresponding command: CTO

Configures the trigger output conditions for the given digital output line. Depending on the controller, the trigger output conditions will either become active immediately, or will become active when activated with PI_TRO().

Arguments:

ID ID of controller

piTriggerOutputIds is an array with the trigger output lines of the controller
piTriggerParameterArray is an array with the CTO parameter IDs
pdValueArray is an array of the values to which the CTO parameters are set
iArraySize is the size of the array *piTriggerOutputIds*

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

```
BOOL PI_CTOSTring (int ID, const int* piTriggerOutputIds, const int*
piTriggerParameterArray, const char* szValueArray, int iArraySize)
```

Corresponding command: CTO

Configures the trigger output conditions for the given digital output line. Depending on the controller, the trigger output conditions will either become active immediately, or will become active when activated with PI_TRO().

Arguments:

ID ID of controller

piTriggerOutputIds is an array with the trigger output lines of the controller

piTriggerParameterArray is an array with the CTO parameter IDs

szValueArray is a list of the values to which the CTO parameters are set. The single values must be separated by a linefeed character

iArraySize is the size of the array *piTriggerOutputIds*

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

```
BOOL PI_CTR (int ID, const char* szAxes, const double* pdValueArray)
```

Corresponding command: CTR

Set relative closed-loop target *for* *szAxes*. Moves the given axes. All axes start moving simultaneously. Servo must be enabled for all commanded axes prior to using this command.

The selected closed-loop control mode (see PI_CMO()) determines the variable which is controlled with PI_CTR() (e.g. position or velocity or force).

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray relative target values for the axes

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

```
BOOL PI_CTV (int ID, const char* szAxes, const double* pdValueArray)
```

Corresponding command: CTV

Set absolute closed-loop target *for* *szAxes*. Moves the given axes. All axes start moving simultaneously. Servo must be enabled for all commanded axes prior to using this command.

The selected closed-loop control mode (see PI_CMO()) determines the variable which is controlled with PI_CTV() (e.g. position or velocity or force).

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray target values for the axes

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_DCO (int *ID*, const char* *szAxes*, const BOOL* *pbValueArray*)**Corresponding command:** DCO

Sets drift compensation mode for given axes (on or off). Drift compensation is applied to avoid unwanted changes in displacement over time and is therefore recommended for static operation. For a detailed description see "Drift Compensation" in the controller User Manual.

Drift compensation is automatically deactivated as long as the wave generator is activated.

Arguments:

ID ID of controller

szAxes string with axes

pbValueArray gives the drift compensation mode, can have the following values:

0 = drift compensation off

1 = drift compensation on

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_DEC (int *ID*, const char* *szAxes*, double* *pdValueArray*)**Corresponding command:** DEC

Set the deceleration to use during moves of *szAxes*. The PI_DEC() setting only takes effect when the given axis is in closed-loop operation (servo on).

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray maximum decelerations for the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_DEL (int *ID*, int *iMilliseconds*)**Corresponding command:** DEL

Delay the controller for *iMilliseconds* milliseconds.

Arguments:

ID ID of controller

iMilliseconds delay value in milliseconds

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_DFH (int *ID*, const char* *szAxes*)**Corresponding command:** DFH

Defines the current positions of *szAxes* as the axis home position (by setting the position value to 0.00).

Arguments:

ID ID of controller

szAxes string with axes, if "" or **NULL** all axes are affected.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_DIO (int *ID*, const int* *piChannelsArray*, const BOOL* *pbValueArray*, int *iArraySize*)

Corresponding command: DIO

Set digital output channels HIGH or LOW.

Arguments:

ID ID of controller

piChannelsArray array containing digital output channel identifiers

pbValueArray array containing the states of specified digital output channels, **TRUE** if HIGH, **FALSE** if LOW

If piChannelsArray contains 0, the array is a bit pattern which gives the states of all lines

iArraySize the size of the array *pbValueArray*

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_DPA (int *ID*, const char* *szPassWord*, const char* *szAxes*, const int* *piParameterArray*)

Corresponding command: DPA

Resets parameters or settings to default values. DPA does not overwrite parameters or parameter-independent settings in non-volatile memory.

Arguments:

ID ID of controller

szPassWord The password depends on the parameter or parameter-independent setting to be reset. See the user manual of the controller for details.

szAxes string with designators. For each designator in szAxes one parameter value is reset.

piParameterArray Array with parameter IDs

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_DRC (int *ID*, const int* *piRecordTableIdsArray*, const char* *szRecordSourceIds*, const int* *piRecordOptionsArray*)

Corresponding command: DRC

Set data recorder configuration: determines the data source (*szRecordSourceIdsArray*) and the kind of data (*piRecordOptionsArray*) used for the given data recorder table.

Arguments:

ID ID of controller

piRecordTableIdsArray ID of the record table

szRecordSourceIds ID of the record source, for example axis number or channel number. The value of this argument depends on the corresponding record option.

piRecordOptionsArray record option, i.e. the kind of data to be recorded

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_DRT (int *ID*, const int* *piRecordTableIdsArray*, const int* *piTriggerSourceArray*, const char* *szValues*, int *iArraySize*)

Corresponding command: DRT

Defines a trigger source for the given data recorder table.

For the data recorder configuration, i.e. for the assignment of data sources and record options to the recorder tables, use PI_DRC().

With PI_qDRR() you can read the last recorded data set.

For more information see "Data Recorder" in the controller User Manual.

Arguments:

ID ID of controller

piRecordTableIdsArray ID of the record table

piTriggerSourceArray ID of the trigger source

szValues depending on the trigger source, value can be a dummy, e.g. an arbitrary character, or the ID of a certain digital input line

iArraySize size of *piRecordTableIdsArray*, *piTriggerSourceArray* and *szValues*.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_EAX (int *ID*, const char* *szAxes*, const BOOL* *pbValueArray*)

Corresponding command: EAX

Enable axis. If disabled, no motion is executed. If motion is commanded for an axis that is not enabled, an error will be set.

Arguments:

ID ID of controller

szAxes string with axes

pbValueArray enable status for the specified axes, **TRUE** for "on", **FALSE** for "off"

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_FDG (int *ID*, const char* *szScanRoutineName*, const char* *szScanAxis*, const char* *szStepAxis*, const char* *szParameters*)

Corresponding command: FDG

Fast alignment: Defines a fast alignment gradient search routine. Use PI_FRS to start the routine. With PI_qFRR(), you can read out the definition and the results of the routine.

See the E712T0016 Technical Note ("Fast Alignment Routines") for detailed descriptions.

Arguments:

ID ID of controller

szScanRoutineName The identifier of the routine.

szScanAxis Identifier of the axis that is to be the master axis of the gradient search routine.

szStepAxis Identifier of the axis that is to be the second axis of the gradient search routine.

szParameters The parameters are optional. For parameters that are omitted, default values will be used.

[ML <stop level>] // float

ML: Required keyword

<stop level>: Gives one stop criterion for the gradient search routine.

[A <alignment signal input channel>] // int

A: Required keyword

<alignment signal input channel>: Identifier of the fast alignment input channel whose maximum intensity is sought.

[MIA <min radius>] // float

MIA: Required keyword

<min radius>: Minimum radius of the circular motion for scan axis and step axis (= amplitude of the sine curve).

[MAA <max radius>] // float

MAA: Required keyword

<max radius>: Maximum radius of the circular motion for scan axis and step axis (= amplitude of the sine curve).

[F <frequency>] // float

F: Required keyword

<frequency>: Frequency of the sine curves for scan axis and step axis.

[SP <speed factor>] // float

SP: Required keyword

<speed factor>: The speed factor can be used to speed up the offset change.

[V <max velocity>] // float
 V: Required keyword
 <max velocity>: Velocity limit for the offset change..
 [MDC <max direction changes>] // int
 MDC: Required keyword
 <max direction changes>: Gives one stop criterion for the gradient search routine.
 [SPO <speed offset>] // float
 SPO: Required keyword
 <speed offset>: Offset that can be applied in the velocity calculation.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_FDR (int *ID*, const char* *szScanRoutineName*, const char* *szScanAxis*, const double *dScanAxisRange*, const char* *szStepAxis*, const double *dStepAxisRange*, const char* *szParameters*)

Corresponding command: FDR

Fast alignment: Defines a fast alignment area scan routine. Use PI_FRS to start the routine. With PI_qFRR(), you can read out the definition and the results of the routine.

See the E712T0016 Technical Note ("Fast Alignment Routines") for detailed descriptions.

Arguments:

ID ID of controller
szScanRoutineName The identifier of the routine.
szScanAxis Identifier of the axis that is to be the master axis of the scan routine.
dScanAxisRange Scan range for the scan axis.
szStepAxis Identifier of the step axis.
dStepAxisRange Scan range for the step axis.
szParameters The parameters are optional. For parameters that are omitted, default values will be used.

[L <threshold level>] // float
 L: Required keyword
 Minimum intensity threshold of the analog input signal. If during an area scan routine no value of the analog input signal is higher than the given minimum threshold level, PI_qFRR() will report "not successful" for the routine.
 [A <alignment signal input channel>] // int
 A: Required keyword
 <alignment signal input channel>: Identifier of the fast alignment input channel whose maximum intensity is sought.
 [F <frequency>] // float
 F: Required keyword
 Frequency of the scan axis.
 [V <velocity>] // float
 V: Required keyword
 <velocity>: Velocity of the step axis.
 [MP1 <scan axis middle position>] // float
 MP1: Required keyword
 <scan axis middle position>: Middle position of the scan range for the scan axis.
 [MP2 <step axis middle position>] // float
 MP2: Required keyword
 <step axis middle position>: Middle position of the scan range for the step axis.
 [TT <target type>] // int
 TT: Required keyword
 <target type>: ID of the area scan type. Possible values:
 0 = raster scan
 1 = spiral scan
 [CM <estimation method>] // int
 CM: Required keyword
 <estimation method>: ID of the estimation method for the position of the global intensity maximum:

0 = global maximum is at the position where the maximum value was recorded
 1 = position of global maximum is calculated from the recorded data using a Gaussian LS fit.
 2 = position of global maximum is calculated from the recorded data using an analogy to a center-of-gravity calculation

[MIIL <minimum level of fast alignment input>] // float

MIIL: Required keyword

<minimum level of fast alignment input>: Minimum intensity to be used for estimation method 1 or 2 (see CM above), in % of the maximum intensity that has been recorded.

[MAIL <maximum level of fast alignment input>] // float

MAIL: Required keyword

<maximum level of fast alignment input>: Maximum intensity to be used for estimation method 1 or 2 (see CM above), in % of the maximum intensity that has been recorded.

[SP <stop position option>] // int

SP: Required keyword

<stop position option>: ID of the position to be approached by scan axis and step axis when the area scan routine has been completed:

0 = move to scan axis and step axis position with the maximum intensity of the analog input signal

1 = stay at the end position of the area scan routine

2 = move to the start position of the area scan routine

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_FED (int *ID*, const char* *szAxes*, const int* *iEdgeArray*, const int* *iParamArray*)

Corresponding command: FED

Moves given axis to a given signal edge and then moves out of any limit condition.

Arguments:

ID ID of controller

szAxes axes to move.

iEdgeArray Defines the type of edge the axis has to move to.

The following edge types are available:

1 = negative limit switch

2 = positive limit switch

3 = reference switch

iParamArray at present, this argument is not needed, should contain zeros

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_FGC(int *ID*, const char* *szProcessIds*, const double* *pdScanAxisCenterValueArray*, const double* *pdStepAxisCenterValueArray*)

Corresponding command: FGC

Fast alignment: Change center position of gradient search routine.

Arguments:

ID ID of controller

szProcessIds The identifier of the routine.

pdScanAxisCenterValueArray Center position of the circular motion for the scan axis.

pdStepAxisCenterValueArray Center position of the circular motion for the step axis.

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_FIO (int *ID*, const char* *szAxis1*, double *dLength1*, const char* *szAxis2*, double *dLength2*, double *dThreshold*, double *dLinearStep*, double *dAngleScan*, int *iAnalogInput*)

Corresponding command: FIO

Starts a scanning procedure for the alignment of optical elements (e.g. optical fibers), the input and output of which are on the same side.

Arguments:

ID ID of controller
szAxis1 first axis that defines scanning area
dLength1 length of scanning area along *szAxis1*
szAxis2 second axis that defines scanning area
dLength2 length of scanning area along *szAxis2*
dThreshold intensity threshold of the analog input signal, in V
dLinearStep step size in which the platform moves along the spiral path
dAngleScan angle around the pivot point at which scanning is done, in degrees
iAnalogInput is the identifier of the analog input signal whose maximum intensity is sought

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_FLM (int *ID*, const char* *szAxis*, double *dLength*, double *dThreshold*, int *iAnalogInput*, int *iDirection*)

Corresponding command: FLM

Starts a scanning procedure to determine the global maximum intensity of an analog input signal.

Arguments:

ID ID of controller
szAxis one axis of the controller, axes X, Y, Z, U, V, W are permissible
dLength distance to be scanned along the axis
dThreshold intensity threshold of the analog input signal, in V
iAnalogInput is the identifier of the analog input signal whose maximum intensity is sought
iDirection indicates the direction of the scanning procedure as well as the starting and end position of the distance

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_FLS (int *ID*, const char* *szAxis*, double *dLength*, double *dThreshold*, int *iAnalogInput*, int *iDirection*)

Corresponding command: FLS

Starts a scanning procedure which scans a specified distance along an axis until the analog input signal reaches a specified intensity threshold.

Arguments:

ID ID of controller
szAxis one axis of the controller, axes X, Y, Z, U, V, W are permissible
dLength distance to be scanned along the axis
dThreshold intensity threshold of the analog input signal, in V
iAnalogInput is the identifier of the analog input signal whose maximum intensity is sought
iDirection indicates the direction of the scanning procedure as well as the starting and end position of the distance

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_FNL (int *ID*, const char* *szAxes*)**Corresponding command:** FNL

Starts a reference move: Moves all axes *szAxes* synchronously to the negative physical limits of their travel ranges and sets the current positions to the negative range limit values.

Note: Call `PI_IsControllerReady()` to find out if referencing is complete (the controller will be "busy" while referencing, so most other commands will cause a `PI_CONTROLLER_BUSY` error) and `PI_qFRF()` to check whether the reference move was successful.

Arguments:

ID ID of controller

szAxes axes to move

Returns:

TRUE if successful, **FALSE** otherwise

Errors:

PI_UNKNOWN_AXIS_IDENTIFIER *cAxis* is not a valid axis identifier

BOOL PI_FPH (int *ID*, const char* *szAxes*)**Corresponding command:** FPH

Starts phase finding process: Finds offset between motor and encoder by performing a homing process.

Notice: The stage will start moving.

Arguments:

ID ID of controller

szAxes axes to perform phase finding

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_FPL (int *ID*, const char* *szAxes*)**Corresponding command:** FPL

Starts a reference move: Moves all axes *szAxes* synchronously to the positive physical limits of their travel ranges and sets the current positions to the positive range limit values.

Note: Call `PI_IsControllerReady()` to find out if referencing is complete (the controller will be "busy" while referencing, so most other commands will cause a `PI_CONTROLLER_BUSY` error) and `PI_qFRF()` to check whether the reference move was successful.

Arguments:

ID ID of controller

szAxes axes to move

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_FRC (int *ID*, const char* *szProcessIdBase*, const char* *szProcessIdsCoupled*)**Corresponding command:** FRC

Fast alignment: Couples fast alignment routines to each other.

Routine types that can be coupled: gradient search routines. Coupled routines are not allowed to stop until all routines coupled to them are finished.

Arguments:

ID ID of controller

szProcessIdBase The identifier of a routine.

szProcessIdsCoupled The identifier of a routine that is to be coupled to the routine given by *szProcessIdBase*

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_FRF (int *ID*, const char* *szAxes*)

Corresponding command: FRF

Starts a reference move: Synchronous reference move of all axes *szAxes*, i.e. the given axis is moved to its physical reference point and the current position is set to the reference position.

Note: Call `PI_IsControllerReady()` to find out if referencing is complete (the controller will be "busy" while referencing, so most other commands will cause a `PI_CONTROLLER_BUSY` error) and `PI_qFRF()` to check whether the reference move was successful.

Arguments:

ID ID of controller
szAxes string with axes

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_FRP (int *ID*, const char* *szScanRoutineNames*, const int* *piOptionsArray*)

Corresponding command: FRP

Fast alignment: Stops, pauses or resumes a fast alignment routine. A routine to be stopped or paused must have been started with `PI_FRS` before. A routine to be resumed with `PI_FRP` must have been paused with `PI_FRP` before.

Arguments:

ID ID of controller
szScanRoutineNames The identifier of the routine.
piOptionsArray The action to be performed for the routine. Possible actions:
 0 = stop the routine
 1 = pause the routine
 2 = resume the routine

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_FRS (int *ID*, const char* *szScanRoutineNames*)

Corresponding command: FRS

Fast alignment: Starts a fast alignment routine. The routine must have been defined before with `PI_FDR()` or `PI_FDG()` or via the appropriate parameters (see E712T0016 Technical Note).

Arguments:

ID ID of controller
szScanRoutineNames The identifier of the routine. Multiple gradient search routines can run synchronously for the axes on both the sender and receiver side. They can be coupled to each other with `PI_FRC()`.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_FSA (int *ID*, const char* *szAxis1*, double *dLength1*, const char* *szAxis2*, double *dLength2*, double *dThreshold*, double *dDistance*, double *dAlignStep*, int *iAnalogInput*)

Corresponding command: FSA

Starts a scanning procedure to determine the maximum intensity of an analog input signal in a plane. The search consists of two subprocedures:

- "Coarse portion"; corresponds to the procedure that is started with the `PI_FSC()` function
- "Fine portion"; corresponds to the procedure that is started with the `PI_AAP()` function

The fine portion is only executed when the coarse portion has previously been successfully completed.

Arguments:**ID** ID of controller**szAxis1** first axis that defines scanning area. Axes X, Y, and Z are permissible. During the coarse portion, the platform is moved in this axis from scanning line to scanning line by the distance given by *dDistance*.**dLength1** length of scanning area along *szAxis1***szAxis2** second axis that defines scanning area. Axes X, Y, and Z are permissible. During the coarse portion, the scanning lines are in this axis.**dLength2** length of scanning area along *szAxis2***dThreshold** intensity threshold of the analog input signal, in V**dDistance** distance between the scanning lines, is only used during the coarse portion**dAlignStep** starting value for the step size, is only used during the fine portion,**iAnalogInput** is the identifier of the analog input signal whose maximum intensity is sought**Returns:****TRUE** if no error, **FALSE** otherwise (see p. 7)
BOOL PI_FSC (int *ID*, const char* *szAxis1*, double *dLength1*, const char* *szAxis2*, double *dLength2*, double *dThreshold*, double *dDistance*, int *iAnalogInput*)
Corresponding command: FSC

Starts a scanning procedure which scans a specified area ("scanning area") until the analog input signal reaches a specified intensity threshold.

The scanning procedure started with PI_FSC() corresponds to the "coarse portion" of the scanning procedure that is started with the PI_FSA function.

Arguments:**ID** ID of controller**szAxis1** the axis in which the platform moves from scanning line to scanning line by the distance given by *dDistance*.**dLength1** length of scanning area along *szAxis1***szAxis2** is the axis in which the scanning lines are located,**dLength2** length of scanning area along *szAxis2***dThreshold** intensity threshold of the analog input signal, in V**dDistance** distance between the scanning lines**iAnalogInput** is the identifier of the analog input signal whose maximum intensity is sought**Returns:****TRUE** if no error, **FALSE** otherwise (see p. 7)
BOOL PI_FSM (int *ID*, const char* *szAxis1*, double *dLength1*, const char* *szAxis2*, double *dLength2*, double *dThreshold*, double *dDistance*, int *iAnalogInput*)
Corresponding command: FSM

Starts a scanning procedure to determine the global maximum intensity of an analog input signal in a plane.

Arguments:**ID** ID of controller**szAxis1** the axis in which the platform moves from scanning line to scanning line by the distance given by *dDistance*.**dLength1** length of scanning area along *szAxis1***szAxis2** is the axis in which the scanning lines are located,**dLength2** length of scanning area along *szAxis2***dThreshold** intensity threshold of the analog input signal, in V**dDistance** distance between the scanning lines**iAnalogInput** is the identifier of the analog input signal whose maximum intensity is sought**Returns:****TRUE** if no error, **FALSE** otherwise (see p. 7)

BOOL PI_GetAsyncBuffer (int *ID*, double ***pnValArray*)

Get address of internal buffer used for storing data read in by a call to PI_qDRR(), PI_qDDL(), PI_qGWD(), PI_qTWS(), PI_qJLT() or PI_qHIT().

Arguments:

ID ID of controller

pnValarray pointer to receive address of internal array used to store the data, the DLL will have allocated enough memory to store all data; call **PI_GetAsyncBufferIndex()** to find out how many data points have been transferred up to that time.

Returns:

TRUE if successful, **FALSE** otherwise

int PI_GetAsyncBufferIndex (int *ID*)

Get index used for the internal buffer filled with data read in by a call to PI_qDRR(), PI_qDDL(), PI_qGWD(), PI_qTWS(), PI_qJLT() or PI_qHIT().

Arguments:

ID ID of controller

Returns:

Index of the data element which was last read in, **-1** otherwise

BOOL PI_GetDynamicMoveBufferSize (int *ID*, long* *pnSize*)**Corresponding command: #11**

#11 gets the free memory space of a buffer that contains the motion profile points. For more information, see "Motions of the Hexapod" in the controller User Manual.

Arguments:

ID ID of controller

pnSize current number of free motion profile points

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_GOH (int *ID*, const char* *szAxes*)**Corresponding command: GOH**

Move all axes in *szAxes* to their home positions (is equivalent to moving the axes to positions 0 using PI_MOV()).

Depending on the controller, the definition of the home position can be changed with PI_DFH().

Arguments:

ID ID of controller

szAxes string with axes, if "" or **NULL** all axes are affected.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_HasPosChanged (int *ID*, const char* *szAxes*, BOOL* *pbValueArray*)**Corresponding command: #6**

Queries whether the axis positions have changed since the last position query was sent.

Arguments:

ID ID of controller

szAxes axis of controller

pbValueArray indicates whether axis positions have changed, the response is bit-mapped

Returns:

TRUE if successful, **FALSE** otherwise (see p. 7)

BOOL PI_HDT (int *ID*, const int* *iDeviceIDsArray*, const int* *iAxisIDsArray*, const int* *piValueArray*, int *iArraySize*)

Corresponding command: HDT

Assigns a lookup table to the given axis of the given HID device.

Arguments:

ID ID of controller

iDeviceIDsArray HID devices connected to the controller

iAxisIDsArray axes of the HID device(s)

piValueArray lookup tables to be assigned. Supported tables depend on the controller.

Possible tables (ID: type):

1: linear

2: parabolic

3: cubic

4: exponential

5: inverted linear

6: inverted parabolic

101 or higher: user-defined tables

iArraySize size of *iDeviceIDsArray*, *iAxisIDsArray* and *piValueArray*

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_HIA (int *ID*, const char* *szAxes*, const int* *iFunctionArray*, const int* *iDeviceIDsArray*, const int* *iAxesIDsArray*)

Corresponding command: HIA

Configures the control of axes of the controller by axes of HID devices ("HID control"): Assigns an axis of an HID device to the given motion parameter of the given axis of the controller.

Arguments:

ID ID of controller

szAxes axes of controller

iFunctionArray motion parameters to be controlled by the axes of HID devices

iDeviceIDsArray HID devices connected to the controller

iAxesIDsArray axes of the HID device(s)

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_HIL (int *ID*, const int* *iDeviceIDsArray*, const int* *iLED_IDsArray*, const int* *pnValueArray*, int *iArraySize*)

Corresponding command: HIL

Sets the current state of the given output unit or characteristic ("LED") of the given HID device.

Arguments:

ID ID of controller

iDeviceIDsArray HID devices connected to the controller

iLED_IDsArray output units or characteristics ("LEDs") of the HID device(s)

pnValueArray states to be set for the output units or characteristics of the HID device(s)

iArraySize size of *iDeviceIDsArray*, *iLED_IDsArray* and *pnValueArray*

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_HIN (int *ID*, const char* *szAxes*, const BOOL* *pbValueArray*)

Corresponding command: HIN

Enables or disables the control by HID devices ("HID control") for the given axis of the controller.

Arguments:

ID ID of controller

szAxes string with axes of the controller

pbValueArray activation state of the HID control for the specified controller axes, **TRUE** for "enabled", **FALSE** for "disabled"

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_HIS (int *ID*, const int* *iDeviceIDsArray*, const int* *iItemIDsArray*, const int* *iPropertyIDArray*, const char* *szValues*, int *iArraySize*)

Corresponding command: HIS

Configures the given HID device.

Arguments:

ID ID of controller

iDeviceIDsArray HID devices connected to the controller

iItemIDsArray operating elements of the HID device(s)

iPropertyIDArray properties of the operating elements of the HID device(s)

szValues string with the values to which the properties of the operating elements are to be set

iArraySize size of *iDeviceIDsArray*, *iItemIDsArray* and *iPropertyIDArray*

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_HIT (int *ID*, const int* *piTableIDsArray*, const int* *piPointNumberArray*, const double* *pdValueArray*, int *iArraySize*)

Corresponding command: HIT

Fills the given lookup table with values.

Arguments:

ID ID of controller

piTableIDsArray lookup tables of the controller

piPointNumberArray points in the lookup table (index begins with 1)

pdValueArray values of the points (range is -1.0 to 1.0)

iArraySize size of *piTableIDsArray*, *piPointNumberArray* and *pdValueArray*

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_HLT (int *ID*, const char* *szAxes*)

Corresponding **command:** HLT

Halt the motion of given axes smoothly.

Error code 10 is set. PI_HLT() does not stop macros.

Arguments:

ID ID of controller

szAxes string with axes, if "" or **NULL** all axes are affected.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_IFC (int *ID*, const char* *szParameters*, const char* *szValues*)

Corresponding command: IFC

Interface configuration.

After PI_IFC() is sent, the new setting becomes active and the host PC interface configuration may need to be changed to maintain communication.

Arguments:

ID ID of controller

szParameters determines which interface <parameter> should be changed. See szValues.

szValues Array with the values of the parameters:

- for **szParameters** = RSBAUD, the **szValues** parameter value gives the baud rate to be used for RS-232 communication
- for **szParameters** = GPADR, the **szValues** parameter value gives the device address to be used for GPIB (IEEE 488) communication
- for **szParameters** = IPADR, the first four portions of the **szValues** parameter value specify the default IP address for TCP/IP communication, the last portion specifies the default port to be used
- for **szParameters** = IPSTART, the **szValues** parameter value defines the startup behavior for configuration of the IP address for TCP/IP communication:
 - 0 = use IP address defined with IPADR
 - 1 = use DHCP to obtain IP address, if this fails, use IPADR
- for **szParameters** = IPMASK, the **szValues** parameter value gives the IP mask to be used for TCP/IP communication, in the form uint.uint.uint.uint

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_IFS (int *ID*, const char* *szPassword*, const char* *szParameters*, const char* *szValues*)

Corresponding command: IFS

Interface parameter store.

The power-on default parameters for the interface are changed in non-volatile memory, but the current active parameters are not. Settings made with PI_IFS() become active with the next power-on or reboot.

Arguments:

ID ID of controller

szPassword > the default password to write to EPROM is 100.

szParameters determines which interface <parameter> should be changed. See szValues.

szValues Array with the values of the parameters:

- for **szParameters** = RSBAUD, the **szValues** parameter value gives the baud rate to be used for RS-232 communication
- for **szParameters** = GPADR, the **szValues** parameter value gives the device address to be used for GPIB (IEEE 488) communication
- for **szParameters** = IPADR, the first four portions of the **szValues** parameter value specify the default IP address for TCP/IP communication, the last portion specifies the default port to be used
- for **szParameters** = IPSTART, the **szValues** parameter value defines the startup behavior for configuration of the IP address for TCP/IP communication:
 - 0 = use IP address defined with IPADR
 - 1 = use DHCP to obtain IP address, if this fails, use IPADR
- for **szParameters** = IPMASK, the **szValues** parameter value gives the IP mask to be used for TCP/IP communication, in the form uint.uint.uint.uint

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_IMP (int *ID*, const char* *szAxes*, const double* *pdImpulseSize*)

Corresponding command: IMP

Starts performing an impulse and recording the impulse response for the given axis. An "impulse" consists of a relative move of the specified amplitude followed by an equal relative move in the opposite direction.

Arguments:

ID ID of controller

szAxes axes for which the impulse response will be recorded

pdImpulseSize array with the pulse height (amplitude values).

Returns:

TRUE if no error **FALSE** otherwise

BOOL PI_IsControllerReady (int *ID*, int * *piControllerReady*)

Corresponding command: #7 (ASCII 7)

Asks controller for ready status (tests if controller is ready to perform a new command).

Arguments:

ID ID of controller

piControllerReady array to receive the status of the controller: 1 if controller is ready

0 if controller is not ready (e.g. performing a referencing command)

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_IsMoving (int *ID*, const char * *szAxes*, BOOL* *pbValueArray*)

Corresponding command: #5 (ASCII 5)

Check if *szAxes* are moving. If an axis is moving the corresponding element of the array will be **TRUE**, otherwise **FALSE**. If no axes were specified, only one boolean value is returned and *pbValueArray[0]* will contain a generalized state: **TRUE** if at least one axis is moving, **FALSE** if no axis is moving.

Arguments:

ID ID of controller

szAxes string with axes, if "" or **NULL** all axes are affected.

pbValueArray array to receive the status of the axes

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_IsRunningMacro (int *ID*, BOOL* *pbRunningMacro*)

Corresponding command: #8 (ASCII 8)

Check if controller is currently running a macro

Arguments:

ID ID of controller

pbRunningMacro pointer to boolean to receive answer: **TRUE** if a macro is running, **FALSE** otherwise

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_JAX (int *ID*, const int *iJoystickID*, const int *iAxesID*, const char* *szAxesBuffer*)

Corresponding command: JAX

Set axis controlled by a joystick which is directly connected to the controller.

Each axis of the controller can only be controlled by one joystick axis.

See "Joystick Control" in the controller User Manual for details.

Arguments:

ID ID of controller

iJoystickID joystick device connected to the controller

iAxesID IDs of the joystick axes

szAxesBuffer name(s) of the axis or axes to be controlled by this joystick axis

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_JDT (int *ID*, const int* *iJoystickIDs*, const int* *iAxesIDs*, const int* *piValarray*, int *iArraySize*)

Corresponding command: JDT

Set default lookup table for the given joystick axis of the given joystick which is directly connected to the controller.

The current valid lookup table for the specified joystick axis is overwritten by the selection made with PI_JDT().

See "Joystick Control" in the controller User Manual for details.

Arguments:

ID ID of controller

iJoystickIDs array with joystick devices connected to the controller

iAxesIDs array with joystick axis to be set

piValarray pointer to array with table types for the corresponding joystick axes, valid table types are:

1 = linear

2 = parabolic

iArraySize size of arrays

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_JLT (int *ID*, int *iJoystickID*, int *iAxisID*, int *iStartAdress*, const double* *pdValueArray*,int *iArraySize*)

Corresponding command: JLT

Fills the lookup table for the given axis of the given joystick device which is connected to the controller.

Arguments:

ID ID of controller

iJoystickID joystick device connected to the controller

iAxisID joystick axis to be set

iStartAdress index of a point in the lookup table, starts with 1

pdValueArray values of the points in the lookup table

iArraySize size of arrays

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_JON (int *ID*, const int* *iJoystickIDs*, const BOOL* *pbValarray*, int *iArraySize*)

Corresponding command: JON

Enable or disable a joystick which is directly connected to the controller.

The joystick must be enabled for joystick control of the controller axis which was assigned to the joystick axis with PI_JAX().

See "Joystick Control" in the controller User Manual for details.

Arguments:

ID ID of controller

iJoystickIDs array with joystick devices connected to the controller

pbValarray pointer to array with joystick enable states (0 for deactivate, 1 for activate)

iArraySize size of arrays

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_KCP (int *ID*, const char* *szSource*, const char* *szDestination*)

Corresponding command: KCP

Copies a coordinate system.

Arguments:

ID ID of controller

szSource name of already defined coordinate system

szDestination name of coordinate system copy

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_KEN (int *ID*, const char* *szNameOfCoordSystem*)

Corresponding command: KEN

Enables an already defined coordinate system.

Arguments:

ID ID of controller

szNameOfCoordSystem name of the coordinate system to be enabled

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_KLD (int *ID*, const char* *szNameOfCoordSystem*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: KLD

Defines a levelling coordinate system (KLD type). A coordinate system defined with KLD is intended to eliminate Hexapod misalignment. Use KLD in case misalignment is known via an external measurable deviation.

Arguments:

ID ID of controller

szNameOfCoordSystem name of the coordinate system to be defined

szAxes string with axes

pdValueArray positions (for axes X, Y, Z) and angles (for axes U, V, W)

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_KLF (int *ID*, const char* *szNameOfCoordSystem*)

Corresponding command: KLF

Defines a levelling coordinate system (KLF type). A coordinate system defined with KLF is intended to eliminate Hexapod misalignment. Use KLF in case the Hexapod is already in the aligned position.

Arguments:

ID ID of controller

szNameOfCoordSystem name of the coordinate system to be defined

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_KLN (int *ID*, const char* *szNameOfChild*, const char* *szNameOfParent*)

Corresponding command: KLN

Links two coordinate systems together by defining a parent-child relation; thus forming a chain.

Arguments:

ID ID of controller

szNameOfChild name of the child coordinate system

szNameOfParent name of the parent coordinate system

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_KRM (int *ID*, const char* *szNameOfCoordSystem*)

Corresponding command: KRM

This command deletes a coordinate system

Arguments:

ID ID of controller

szNameOfCoordSystem name of the coordinate system to be deleted

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_KSB (int *ID*, const char* *szNameOfCoordSystem*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: KSB

Defines a new coordinate system of KSB type by changing the orientation of the base coordinate system (possible in steps of 90°).

Arguments:

ID ID of controller

szNameOfCoordSystem name of the coordinate system to be defined

szAxes string with axes, possible values are U, V, W

pdValueArray angles in degrees, possible values are 0, 90, 180, 270, -90, -180, -270

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_KSD (int *ID*, const char* *szNameOfCoordSystem*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: KSD

This command defines a new KSD type coordinate system. The center of rotation can be changed with the coordinates X, Y, Z. The orientation of the coordinate system can be changed with the coordinates U, V, W.

Arguments:

ID ID of controller

szNameOfCoordSystem name of the coordinate system to be defined

szAxes string with axes

pdValueArray positions (for axes X, Y, Z) and angles (for axes U, V, W)

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_KSF (int *ID*, const char* *szNameOfCoordSystem*)

Corresponding command: KSF

This command defines a new KSF type coordinate system based on the current position and orientation of the Hexapod platform.

Arguments:

ID ID of controller

szNameOfCoordSystem name of the coordinate system to be defined

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_KST (int *ID*, const char* *szNameOfCoordSystem*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: KST

This command defines a new Tool coordinate system (KST type).

Arguments:

ID ID of controller

szNameOfCoordSystem name of the coordinate system to be defined

szAxes string with axes

pdValueArray positions (for axes X, Y, Z) and angles (for axes U, V, W)

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_KSW (int *ID*, const char* *szNameOfCoordSystem*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: KSW

This command defines a new Work coordinate system (KSW type).

Arguments:

ID ID of controller

szNameOfCoordSystem name of the coordinate system to be defined

szAxes string with axes

pdValueArray positions (for axes X, Y, Z) and angles (for axes U, V, W)

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_MAC_BEG (int *ID*, const char * *szMacroName*)

Corresponding command: MAC BEG

Put the DLL in macro recording mode. This function sets a flag in the library and effects the operation of other functions. Function will fail if already in recording mode. If successful, the commands that follow become part of the macro, so do not check error state unless FALSE is returned. End the recording with PI_MAC_END().

See "Controller Macros" and the MAC command description in the controller User Manual for details.

Arguments:

ID ID of controller

szMacroName name under which macro will be stored in the controller

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_MAC_DEF (int *ID*, const char * *szMacroName*)

Corresponding command: MAC DEF

Set macro with name *szMacroName* as start-up macro. This macro will be automatically executed with the next power-on or reboot of the controller. If *szMacroName* is omitted, the current start-up macro selection is canceled. To find out what macros are available call PI_qMAC().

See "Controller Macros" and the MAC command description in the controller User Manual for details.

Arguments:

ID ID of controller

szMacroName name of the macro to be the start-up macro

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_MAC_DEL (int *ID*, const char * *szMacroName*)**Corresponding command:** MAC DELDelete macro with name *szMacroName*. To find out what macros are available call PI_qMAC().

See "Controller Macros" and the MAC command description in the controller User Manual for details.

Arguments:*ID* ID of controller*szMacroName* name of the macro to delete**Returns:****TRUE** if successful, **FALSE** otherwise**BOOL PI_MAC_END** (int *ID*)**Corresponding command:** MAC END

Take the DLL out of macro recording mode. This function resets a flag in the library and effects the operation of certain other functions. Function will fail if the DLL is not in recording mode.

See "Controller Macros" and the MAC command description in the controller User Manual for details.

Arguments:*ID* ID of controller**Returns:****TRUE** if successful, **FALSE** otherwise**BOOL PI_MAC_NSTART** (int *ID*, const char * *szMacroName*, int *nrRuns*)**Corresponding command:** MAC NSTARTStart macro with name *szMacroName*. The macro is repeated *nrRuns* times. To find out what macros are available call PI_qMAC().

See "Controller Macros" and the MAC command description in the controller User Manual for details.

Arguments:*ID* ID of controller*szMacroName* string with name of the macro to start*nrRuns* number of runs**Returns:****TRUE** if successful, **FALSE** otherwise**BOOL PI_MAC_NSTART_Args** (int *ID*, const char * *szMacroName*, int *nrRuns*, const char * *szArgs*)**Corresponding command:** MAC NSTARTStart macro with name *szMacroName*. The macro is repeated *nrRuns* times. Another execution is started when the last one is finished.*szArgs* stands for the value of a local variable contained in the macro. The sequence of the values in the input must correspond to the numbering of the appropriate local variables, starting with the value of the local variable 1. The individual values must be separated from each other with spaces. A maximum of 256 characters are permitted per function line. *szArgs* can be given directly or via the value of another variable.

To find out what macros are available call PI_qMAC().

See "Controller Macros" and the MAC command description in the controller User Manual for details.

Arguments:*ID* ID of controller*szMacroName* string with name of the macro to start*nrRuns* number of runs*szArgs* value of a local variable contained in the macro**Returns:****TRUE** if successful, **FALSE** otherwise

BOOL PI_MAC_qDEF (int *ID*, char * *szBuffer*, int *iBufferSize*)**Corresponding command:** MAC DEF?

Ask for the start-up macro.

See "Controller Macros" and the MAC command description in the controller User Manual for details.

Arguments:*ID* ID of controller*szBuffer* buffer to receive the string read in from controller, contains the name of the start-up macro. If no start-up macro is defined, the response is an empty string with the terminating character.*iBufferSize* size of *szBuffer*, must be given to avoid buffer overflow.**Returns:****TRUE** if successful, **FALSE** otherwise**BOOL PI_MAC_qERR** (int *ID*, char * *szBuffer*, int *iBufferSize*)**Corresponding command:** MAC ERR?

Reports the last error which occurred during macro execution.

See "Controller Macros" and the MAC command description in the controller User Manual for details.

Arguments:*ID* ID of controller*szBuffer* buffer to receive the string read in from controller, contains the error code number.*iBufferSize* size of *szBuffer*, must be given to avoid buffer overflow.**Returns:****TRUE** if successful, **FALSE** otherwise.**BOOL PI_MAC_qFREE** (int *ID*, int * *pnFreeSpace*)**Corresponding command:** MAC ERR?

Gets the free memory space for macro recording.

See "Controller Macros" and the MAC command description in the controller User Manual for details.

Arguments:*ID* ID of controller*pnFreeSpace* number of characters in bytes for which free memory is still available.**Returns:****TRUE** if successful, **FALSE** otherwise.**BOOL PI_MAC_START** (int *ID*, const char * *szMacroName*)**Corresponding command:** MAC STARTStart macro with name *szMacroName*. To find out what macros are available call PI_qMAC().

See "Controller Macros" and the MAC command description in the controller User Manual for details.

Arguments:*ID* ID of controller*szMacroName* string with name of the macro to start**Returns:****TRUE** if successful, **FALSE** otherwise**BOOL PI_MAC_START_Args** (int *ID*, const char * *szMacroName*, const char* *szArgs*)**Corresponding command:** MAC STARTStart macro with name *szName*. *szArgs* has the same function as with PI_MAC_NSTART_Args.

To find out what macros are available call PI_qMAC().

See "Controller Macros" and the MAC command description in the controller User Manual for details.

Arguments:

ID ID of controller
szMacroName string with name of the macro to start
szArgs value of a local variable contained in the macro

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_MEX (int *ID*, const char * *szCondition*)

Corresponding command: MEX

Stop macro execution due to a given condition of the following type: a specified value is compared with a queried value according to a specified rule.

Can only be used in macros.

When the macro interpreter accesses this command the condition is checked. If it is true the current macro is stopped, otherwise macro execution is continued with the next line. Should the condition be fulfilled later, the interpreter will ignore it.

See also PI_WAC().

See "Controller Macros" and the MEX command description in the controller User Manual for details.

Arguments:

ID ID of controller
szCondition string with condition to evaluate **Returns:**
TRUE if successful, **FALSE** otherwise

BOOL PI_MOD (int *ID*, const char* *szItems*, const unsigned int * *iModeArray*, const char* *szValues*)

Corresponding command: MOD

Set modes for axes / channels / system.

Arguments:

ID ID of controller
szItems string with item identifiers
iModeArray array with IDs of modes to be set
zsValues string with values for each mode to be set

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_MOV (int *ID*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: MOV

Move *szAxes* to specified absolute positions. Axes will start moving to the new positions if ALL given targets are within the allowed ranges and ALL axes can move. All axes start moving simultaneously. Servo must be enabled for all commanded axes prior to using this command.

Arguments:

ID ID of controller
szAxes string with axes
pdValueArray target positions for the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_MRT (int *ID*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: MRT

Move *szAxes* relative to current position and orientation in Tool coordinate system. Position and orientation of the Tool coordinate system change with each motion of the platform.

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray amounts to be added (algebraically) to current target positions of the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_MRW (int *ID*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: MRW

Move *szAxes* relative to current position and orientation in Work coordinate system. Position and orientation of the Work coordinate system do NOT change with motions of the platform.

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray amounts to be added (algebraically) to current target positions of the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_MVE (int *ID*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: MVE

Set new absolute target positions for given axes. Axes will start moving to the new positions if ALL given targets are within the allowed range and ALL axes can move.

If the affected axes are mounted in a way that they move perpendicular to each other, the combined motion of them will describe a linear path. This is achieved by appropriate calculation of accelerations, velocities and decelerations. The current settings for velocity, acceleration and deceleration define the maximum possible values, and the slowest axis determines the resulting velocities.

All axes start moving simultaneously.

This command can be interrupted by PI_STP() and PI_HLT(). No other motion commands (e.g. PI_MOV(), PI_MVR()) are allowed during vector move.

Servo must be enabled for all commanded axes prior to using this command. If servo is switched off or motion error occurs during motion, all axes are stopped.

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray target positions for the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_MVR (int *ID*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: MVR

Move *szAxes* relative to current target position. The new target position is calculated by adding the given position value to the last commanded target value. Axes will start moving to the new position if ALL given targets are within the allowed range and ALL axes can move. All axes start moving simultaneously. Servo must be enabled for all commanded axes prior to using this command.

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray amounts to be added (algebraically) to current target positions of the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_NAV (int *ID*, const int* *piAnalogChannelIds*, const int* *piNrReadingsValues*, int *iArraySize*)

Corresponding command: NAV

Determines the number of readout values of the analog input that are averaged.

Arguments:

ID ID of controller

piAnalogChannelIds identifier of the analog input channel

piNrReadingsValues number of readout values of the analog signal

iArraySize size of arrays *piAnalogChannelIds* and *piNrReadingsValues*

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_NLM (int *ID*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: NLM

Set lower limits ("soft limit") for the positions of *szAxes*.

Depending on the controller, the soft limits are activated and deactivated with PI_SSL().

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray lower limits for position

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_OAC (int *ID*, const int* *piPIEZOWALKChannelsArray*, const double* *pdValueArray*, int *iArraySize*);

Corresponding command: OAC

Set open-loop acceleration of *szAxes*. The PI_OAC setting only takes effect when the given axis is in open-loop operation (servo off).

Arguments:

ID ID of controller

piPIEZOWALKChannelsArray array with PiezoWalk channels

pdValueArray acceleration value

iArraySize the size of the arrays with the PiezoWalk channels and acceleration values

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_OAD (int *ID*, const int* *piPIEZOWALKChannelsArray*, const double* *pdValueArray*, int *iArraySize*)

Corresponding command: OAD

Open-loop analog driving of the given PiezoWalk channel.

Servo must be disabled for the commanded axis prior to using this command (open-loop operation).

Arguments:

ID ID of controller

piPIEZOWALKChannelsArray array with PiezoWalk channels

pdValueArray is the feed voltage amplitude in V, see above for details

iArraySize the size of the arrays with the PiezoWalk channels and feed voltages

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

```

BOOL PI_ODC (int ID, const int* piPIEZOWALKChannelsArray, const double*
pdValueArray, int iArraySize);

```

Corresponding command: ODC

Set open-loop deceleration of *szAxes*. The PI_ODC setting only takes effect when the given axis is in open-loop operation (servo off).

Arguments:

ID ID of controller

piPIEZOWALKChannelsArray array with PiezoWalk channels

pdValueArray deceleration value

iArraySize the size of the arrays with the PiezoWalk channels and deceleration values

Returns:

TRUE if successful, **FALSE** otherwise

```

BOOL PI_OMA (int ID, const char* szAxes, const double* pdValueArray);

```

Corresponding command: OMA

Commands *szAxes* to the given absolute position. Motion is realized in open-loop nanostepping mode.

Servo must be disabled for the commanded axis prior to using this function (open-loop operation).

With PI_OMA() there is no position control (i.e. the target position is not maintained by any control loop).

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray target positions for the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

```

BOOL PI_OMR (int ID, const char* szAxes, const double* pdValueArray);

```

Corresponding command: OMR

Commands *szAxes* to a position relative to the last commanded open-loop target position. The new open-loop target position is calculated by adding the given value *pdValueArray* to the last commanded target value. Motion is realized in nanostepping mode. Servo must be disabled for the commanded axis prior to using this function (open-loop operation). With PI_OMR there is no position control (i.e. the target position is not maintained by a control loop).

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray target positions for the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

```

BOOL PI_ONL (int ID, const int* iPiezoChannels, const int* pdValarray, int
iArraySize)

```

Corresponding command: ONL

Sets control mode for given piezo channel (ONLINE or OFFLINE mode).

Arguments:

ID ID of controller

iPiezoChannels string with piezo channels

pdValueArray gives the control mode, can have the following values:

0 = OFFLINE mode, the output voltage depends on analog control input and DC offset applied to the channel

1 = ONLINE mode, the controller controls the generation of the output voltage

In ONLINE mode the SERVO switches of all channels must be set to OFF on the piezo control electronics.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_OSM (int *ID*, const int* *piPIEZOWALKChannelsArray*, const int* *piValueArray*, int *iArraySize*)

Corresponding command: OSM

Open-loop step moving of the given PiezoWalk channel.

Prior to using PI_OSM(), servo must be disabled for the axis to which the PiezoWalk channel is assigned (open-loop operation).

Arguments:

ID ID of controller

piPIEZOWALKChannelsArray array with PiezoWalk channels

piValueArray number of steps for the PiezoWalk channels (integer steps only)

iArraySize the size of the arrays with the PiezoWalk channels and number of steps

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_OSMf (int *ID*, const int* *piPIEZOWALKChannelsArray*, const double* *pdValueArray*, int *iArraySize*)

Corresponding command: OSM

Open-loop step moving of the given PiezoWalk channel.

Prior to using PI_OSMf(), servo must be disabled for the axis to which the PiezoWalk channel is assigned (open-loop operation).

PI_OSMf() is identical with PI_OSM() but allows to command parts of a step cycle (floating-point numbers are accepted).

Arguments:

ID ID of controller

piPIEZOWALKChannelsArray array with PiezoWalk channels

pdValueArray number of steps for the PiezoWalk channels (floating-point numbers)

iArraySize the size of the arrays with the PiezoWalk channels and number of steps

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_OVL (int *ID*, const int* *piPIEZOWALKChannelsArray*, double* *pdValueArray*, int *ArraySize*)

Corresponding command: OVL

Set velocity for open-loop nanostepping motion of given PiezoWalk channel.

The PI_OVL() setting only takes effect when the given axis is in open-loop operation (servo off).

Arguments:

ID ID of controller

piPIEZOWALKChannelsArray Array with PIEZOWALK channels

pdValueArray maximum velocities for the axes

iArraySize number of items in arrays

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_PLM (int *ID*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: PLM

Set upper limits ("soft limit") for the positions of *szAxes*.

Depending on the controller, the soft limits are activated and deactivated with PI_SSL().

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray upper limits for position

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_POS (int *ID*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: POS

Set current position for given axis (does not cause motion). An axis is considered as "referenced" when the position was set with PI_POS(), so that PI_qFRF() replies "1". Setting the current position with PI_POS() is only possible when the referencing mode is set to "0", see PI_RON().

CAUTION:

The "software-based" travel range limits (PI_qTMN() and PI_qTMX()) and the "software-based" home position (PI_qDHF()) are not adapted when a position value is set with PI_POS(). This may result in

- target positions which are inside the range limits but can not be reached by the hardware—the mechanics is at the hardstop but tries to move further and must be stopped with PI_STP()
- target positions which can be reached by the hardware but are outside of the range limits—e.g. the mechanics is at the negative hardstop and physically could move to the positive hardstop, but due to the software based-travel range limits the target position is not accepted and no motion is possible
- a home position which is outside of the travel range.

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray new axis positions in physical units

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_qACC (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: ACC?

Gets the acceleration value set with PI_ACC() for closed-loop operation.

Arguments:

ID ID of controller

szAxes string with axes, if "" or NULL all axes are queried.

pdValueArray array to be filled with the acceleration settings of the axes

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_qAOS (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: AOS?

Get Analog Input Offset, which was set by PI_AOS() or by a parameter command.

Arguments:

ID ID of controller

szAxes string with axes, if "" or NULL all axes are queried.

pdValueArray array to be filled with analog offset of the axes

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_qATC (int *ID*, const int* *piChannels*, int* *piValueArray*, int *iArraySize*)

Corresponding command: ATC?

Get the options used for the latest auto calibration procedure started with PI_ATC().

See "Calibration Settings" in the User Manual of the controller for more information.

Arguments:

ID ID of controller

piChannels string with channels of the piezo control electronics

piValueArray comprises the settings of the latest auto calibration procedure.

iArraySize size of arrays

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_qATZ (int *ID*, const char* *szAxes*, int* *piAtzResult*)

Corresponding command: ATZ?

Reports if the AutoZero procedure called by PI_ATZ() was successful

Arguments:

ID ID of controller

szAxes string with axes, if "" or NULL all axes are queried.

piAtzResult 1 if PI_ATZ was successful performed, 0 if not successful

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_qATS (int *ID*, const int* *piChannels*, const int* *piOptions*, int* *piValueArray*, int *iArraySize*)

Corresponding command: ATS?

Query the results of the latest auto calibration procedure started with PI_ATC().

See "Calibration Settings" in the User Manual of the controller for more information.

Arguments:

ID ID of controller

piChannels string with channels of the piezo control electronics

piOptions gives the option to be queried. See PI_ATC() for details.

piValueArray gives the results of the latest auto calibration procedure. If 0, the PI_ATC() procedure was successful. Values >0 indicate option specific error codes; multiple non-zero error codes for the same channel and option will be listed one after another.

iArraySize size of arrays

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_qBRA (const int *ID*, char * *szBuffer*, const int *maxlen*)

Corresponding command: BRA?

Gets brake activation state of given axes.

Arguments:

ID ID of controller

szBuffer buffer to store the read in string

maxlen size of *buffer*, must be given to avoid a buffer overflow.

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_qCAV (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: CAV?

Get the current value of the variable controlled by the selected closed-loop control mode (see PI_CMO() for selection).

Arguments:

ID ID of controller

szAxes string with axes, if "" or **NULL** all axes are queried.

pdValueArray array to be filled with current values of the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qCCL (int *ID*, int* *piCommandLevel*)

Corresponding command: CCL?

Returns the current *CommandLevel*.

Arguments:

ID ID of controller

piCommandLevel variable to receive the current command level. See PI_CCL() for possible values.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qCCV (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: CCV?

Get currently valid control value.

PI_qCCV() queries the control value in open-loop and closed-loop operation.

Arguments:

ID ID of controller

szAxes string with axes, if "" or **NULL** all axes are queried.

pdValueArray array to be filled with current control values of the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qCMN (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: CMN?

Get the minimum commandable closed-loop target of *szAxes*. The physical unit and hence the interpretation of the value depend on the closed-loop control mode which is selected for the axis (see PI_CMO() for selection).

Arguments:

ID ID of controller

szAxes string with axes, if "" or **NULL** all axes are queried.

pdValueArray array to receive the minimum commandable closed-loop target of the axes in physical units.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qCMO (int *ID*, const char* *szAxes*, int* *piValueArray*)

Corresponding command: CMO?

Get the closed-loop control mode which is currently selected for *szAxes*..

ID ID of controller

szAxes string with axes, if "" or **NULL** all axes are queried

piValueArray array to receive modes for the specified axes

Returns:

TRUE if successful, **FALSE** otherwise (see p. 7)

BOOL PI_qCMX (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: CMX?

Get the maximum commandable closed-loop target of *szAxes*. The physical unit and hence the interpretation of the value depend on the closed-loop control mode which is selected for the axis (see PI_CMO() for selection).

Arguments:

ID ID of controller

szAxes string with axes, if "" or **NULL** all axes are queried

pdValueArray array to receive the maximum commandable closed-loop target of the axes in physical units.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qCOV (int *ID*, const int* *piChannelsArray*, double* *pdValueArray*, int *iArraySize*)

Corresponding command: COV?

Get current open-loop velocity

Arguments:

ID ID of controller

piChannelsArray is an array with the channels to be queried

pdValueArray array to receive the values

iArraySize is the size of the array **pdValueArray**

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qCST (int *ID*, const char* *szAxes*, char* *szNames*, int *iBufferSize*)

Corresponding command: CST?

Get the type names of the stages associated with *szAxes*. The individual names are preceded by the one-character axis identifier followed by "=" the stage name and a "\n" (line-feed). The line-feed is preceded by a space on every line except the last.

Arguments:

ID ID of controller

szAxes identifiers of the axes, if "" or **NULL** all axes are queried

szNames buffer to receive the string read in from controller, lines are separated by '\n' ("line-feed")

iBufferSize size of *szNames*, must be given to avoid a buffer overflow.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qCSV (int *ID*, double* *pdCommandSyntaxVersion*)

Corresponding command: CSV?

Returns the current *CommandSyntaxVersion*.

Arguments:

ID ID of controller

pdCommandSyntaxVersion variable to receive the current command syntax version (2.0 for GCS 2.0).

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)


```

BOOL PI_qCTI (int ID, const int* piTriggerInputIds, const int*
piTriggerParameterArray, char* szValueArray, int iArraySize, int iBufferSize)

```

Corresponding command: CTI?

Get the trigger input configuration for the given trigger input line.

Arguments:

ID ID of controller

piTriggerInputIds is an array with the trigger input lines of the controller

piTriggerParameterArray is an array with the CTI parameter IDs

szValueArray buffer to receive the values to which the CTI parameters are set, each line has a value of a single CTI parameter, lines are separated by '\n' ("line-feed")

iArraySize is the size of the array *piTriggerInputIds*

iBufferSize size of *szValueArray*, must be given to avoid a buffer overflow.

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

```

BOOL PI_qCTO (int ID, const int* piTriggerOutputIds, const int*
piTriggerParameterArray, double* pdValueArray, int iArraySize)

```

Corresponding command: CTO?

Get the trigger output configuration for the given trigger output line.

Arguments:

ID ID of controller

piTriggerOutputIds is an array with the trigger output lines of the controller

piTriggerParameterArray is an array with the CTO parameter IDs

pdValueArray buffer to receive the values to which the CTO parameters are set

iArraySize is the size of the array *piTriggerOutputIds*

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

```

BOOL PI_qCTOString (int ID, const int* piTriggerOutputIds, const int*
piTriggerParameterArray, char* szValueArray, int iArraySize, int iBufferSize)

```

Corresponding command: CTO?

Get the trigger output configuration for the given trigger output line.

Arguments:

ID ID of controller

piTriggerOutputIds is an array with the trigger output lines of the controller

piTriggerParameterArray is an array with the CTO parameter IDs

szValueArray buffer to receive the values to which the CTO parameters are set, each line has a value of a single CTO parameter, lines are separated by '\n' ("line-feed")

iArraySize is the size of the array *piTriggerOutputIds*

iBufferSize size of *szValueArray*, must be given to avoid a buffer overflow.

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

```

BOOL PI_qCTV (int ID, const char* szAxes, double* pdValueArray)

```

Corresponding command: CTV?

Get the currently valid closed-loop target for *szAxes*. The physical unit and hence the interpretation of the value depend on the closed-loop control mode which is selected for the axis (see PI_CMO() for selection).

Use PI_qCAV() to get the current value of the controlled variable.

Arguments:

ID ID of controller

szAxes string with axes, if "" or **NULL** all axes are queried.

pdValueArray array to be filled with target values of the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qDCO (int *ID*, const char* *szAxes*, BOOL* *pbValueArray*)

Corresponding command: DCO?

Gets drift compensation mode of *szAxes*

Arguments:

ID ID of controller

szAxes string with axes, if "" or **NULL** all axes are queried

pbValueArray array to receive the drift compensation modes of the specified axes, **TRUE** for "on", **FALSE** for "off"

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qDEC (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: DEC?

Gets the deceleration value for closed-loop operation set with PI_DEC().

Arguments:

ID ID of controller

szAxes string with axes, if "" or **NULL** all axes are queried.

pdValueArray array to be filled with the deceleration settings of the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qDFH (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: DFH?

Returns the sensor positions the current home position definitions of *szAxes* are based on.

Arguments:

ID ID of controller

szAxes string with axes, if "" or **NULL** all axes are queried.

pdValueArray array to receive the sensor positions that were valid when PI_DFH() was called the last time (are used as offsets for the calculation of the current axis positions)

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qDIO (int *ID*, const long* *piChannelsArray*, BOOL* *pbValueArray*, int *iArraySize*)

Corresponding command: DIO?

Returns the states of the specified digital input channels.

Use PI_qTIO() (p. 90) to get the number of installed digital I/O channels.

Arguments:

ID ID of controller

piChannelsArray array containing digital output channel identifiers

pbValueArray array containing the states of specified digital output channels, **TRUE** if HIGH, **FALSE** if LOW

Depending on the controller, *piChannelsArray* can contain 0. In this case, the array is a bit pattern which gives the states of all lines.

iArraySize the size of *piChannelsArray* and *pbValueArray*

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qDRC (int *ID*, const int* *piRecordTableIdsArray*, char* *szRecordSourceIds*, int* *piRecordOptionArray*, int *iRecordSourceIdsBufferSize*, int *iRecordOptionArraySize*)

Corresponding command: DRC?

Returns the data recorder configuration for the queried record table. The configuration can be changed with PI_DRC(). The recorded data can be read with PI_qDRR().

Trigger options for recording can be set with PI_DRT() and read with PI_qDRT().

Arguments:

ID ID of controller

piRecordTableIdsArray array of the record table IDs.

szRecordSourceIds array to receive the record source (for example axis number or channel number. The meaning of this value depends on the corresponding record option).

piRecordOptionsArray array to receive the record option, i.e. the kind of data to be recorded

iRecordSourceIdsBufferSize size of *szRecordSourceIds*, must be given to avoid a buffer overflow

iRecordOptionArraySize size of *piRecordTableIdsArray* and *piRecordOptionsArray*

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qDRL (int *ID*, const int* *piRecordTableIdsArray*, int* *piNumberOfRecordedValuesArray*, int *iArraySize*)

Corresponding command: DRL?

Reads the number of points comprised by the last recording, i.e. the number of values that have been recorded since data recording was last triggered. This way it is possible to find out if recording has been finished (all desired points are in the record table) or to find out how many points can be currently read from the record table. Depending on the controller, reading more points than the number returned by PI_qDRL can also read old record table content.

Arguments:

ID ID of controller

piRecordTableIdsArray array of the record channel IDs

piNumberOfRecordedValuesArray array to receive the number of values that have been recorded since recording was last triggered or PI_DRC() was called for the record channel

iArraySize the size of the arrays *piRecordTableIdsArray*, *piNumberOfRecordedValuesArray*

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qDRR (int *ID*, const int* *piRecTableIdsArray*, int *iNumberOfRecTables*, int *iOffsetOfFirstPointInRecordTable*, int *iNumberOfValues*, double** *pdValueArray*, char* *szGcsArrayHeader*, int *iGcsArrayHeaderMaxSize*)

Corresponding command: DRR?

Read data record tables. This function reads the data asynchronously, it will return as soon as the data header has been read and start a background process which reads in the data itself. See PI_GetAsyncBuffer() and PI_GetAsyncBufferIndex(). Detailed information about the data read in can be found in the header sent by the controller. See the GCS Array manual for details.

It is possible to read the data while recording is still in progress.

The data is stored on the controller only until a new recording is done or the controller is powered down.

For more information see "Data Recorder" in the controller User Manual.

Arguments:

ID ID of controller

piRecTableIdArray IDs of data record tables

iNumberOfRecTables number of record tables to read
iOffsetOfFirstPointInRecordTable index of first value to be read (starts with index 1)
iNumberOfValues number of values to read
pdValarray pointer to internal array to store the data; data from all tables read will be placed in the same array with the values interspersed; the DLL will allocate enough memory to store all data, call ***PI_GetAsyncBufferIndex()*** to find out how many data points have already been transferred
szGcsArrayHeader buffer to store the GCS array header
iGcsArrayHeaderMaxSize size of the buffer to store the GCS Array header, must be given to prevent buffer overflow

Returns:

TRUE if successful, **FALSE** otherwise

BOOL *PI_qDRR_SYNC* (int *ID*, int *iRecordTableId*, int *iOffsetOfFirstPointInRecordTable*, int *iNumberOfValues*, double* *pdValueArray*)

Corresponding command: DRR?

Returns the data points of the last recorded data set.

It is possible to read the data while recording is still in progress.

The data is stored on the controller only until a new recording is done or the controller is powered down.

For detailed information see "Data Recorder" in the controller User Manual.

Arguments:

ID ID of controller

iRecordTableId Id of the record table.

iOffsetOfFirstPointInRecordTable The start point in the specified record table (starts with index 1)

iNumberOfValues The number of values to read.

pdValueArray array to receive the values

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL *PI_qDRT* (int *ID*, const int* *piRecordTableIdsArray*, int* *piTriggerSourceArray*, char* *szValues*, int *iArraySize*, int *iValueBufferLength*)

Corresponding command: DRT?

Returns the current trigger source setting for the given data recorder table.

Arguments:

ID ID of controller

piRecordTableIdsArray array of the record table IDs

piTriggerSourceArray array to receive the trigger source

szValues buffer to receive the trigger-source-dependent value

iArraySize size of *piRecordTableIdsArray* and *piTriggerSourceArray*

iValueBufferLength size of *szValues*, must be given to prevent buffer overflow

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL *PI_qEAX* (int *ID*, const char* *szAxes*, BOOL* *pbValueArray*)

Corresponding command: EAX?

Get enable status of axes: enabled/not enabled.

Arguments:

ID ID of controller

szAxes string with axes

pbValueArray enable status for the specified axes, **TRUE** for "enabled", **FALSE** for "not enabled"

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qECO (int *ID*, const char* *szSendString*, char* *szValues*, int *iBufferSize*)

Corresponding command: ECO?

Returns a string. PI_qECO() can be used to test the communication.

Arguments:

ID ID of controller

szSendString array of any given combination of characters consisting of letters and numbers

szValues buffer to receive the string read in from controller

iBufferSize size of *szValues*

Returns:

TRUE if query successful, **FALSE** otherwise

BOOL PI_qERR (int *ID*, long* *pnError*)

Corresponding command: ERR?

Get the error state of the controller. Because the DLL may have queried (and cleared) controller error conditions on its own, it is safer to call PI_GetError() which will first check the internal error state of the library. For a list of possible error codes see p. 123.

Arguments:

ID ID of controller

pnError integer to receive error code of the controller

Returns:

TRUE if query successful, **FALSE** otherwise

BOOL PI_qFGC(int *ID*, const char* *szProcessIds*, double* *pdScanAxisCenterValueArray*, double* *pdStepAxisCenterValueArray*)

Corresponding command: FGC?

Fast alignment: Gets the current center position of the circular motion of a gradient search routine.

Arguments:

ID ID of controller

szProcessIds The identifier of the routine.

pdScanAxisCenterValueArray Current center position of the circular motion for the scan axis

pdStepAxisCenterValueArray Current center position of the circular motion for the step axis

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qFPH (int *ID*, const char* *szAxes*, BOOL* *pbValueArray*)

Corresponding command: FPH?

Gets found phase (offset between motor and encoder) for *szAxes*.

Arguments:

ID ID of controller

szAxes string with axes

pbValueArray array to receive the phase offsets found for the specified axes, **TRUE** for "on", **FALSE** for "off"

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qFRC (int *ID*, const char* *szProcessIdsBase*, char* *szBuffer*, int *iBufferSize*)

Corresponding command: FRC?

Fast alignment: Gets coupled fast alignment routines.

Arguments:

ID ID of controller

szProcessIdBase The identifier of the routine to be queried.

szBuffer buffer to receive the string read in from controller, lines are separated by '\n' ("line-feed").
Contains the identifiers of routines that are coupled to the routine given by *szProcessIdBase*.

iBufferSize size of *szBuffer*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qFRF (int *ID*, const char* *szAxes*, BOOL* *pbValueArray*)

Corresponding command: FRF?

Indicates whether the given axis is referenced or not.

An axis is considered as "referenced" when the current position value is set to a known position.

Depending on the controller, this is the case if a reference move was successfully executed with PI_FRF(), PI_FNL() or PI_FPL(), or if the position was set manually with PI_POS().

Arguments:

ID ID of controller

szAxes string with axes, if "" or **NULL** all axes are affected.

pbValueArray array to receive, 1 if successful, 0 if axis is not referenced (e.g. referencing move failed or has not finished yet)

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qFRH (int *ID*, char* *szBuffer*, int *iBufferSize*)

Corresponding command: FRH?

Fast alignment: Lists descriptions and physical units for the routine results that can be queried with PI_qFRR().

Arguments:

ID ID of controller

szBuffer buffer to receive the string read in from controller, lines are separated by '\n' ("line-feed")

iBufferSize size of *szBuffer*, must be given to avoid buffer overflow

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qFRP (int *ID*, const char* *szScanRoutineNames*, int* *piOptionsArray*)

Corresponding command: FRP?

Fast alignment: Gets the current state of a fast alignment routine.

Arguments:

ID ID of controller

szScanRoutineNames The identifier of the routine.

piOptionsArray is the current state of the routine. Possible states:

0 = routine has been stopped / is not running

1 = routine has been paused

2 = routine is running

If no routine ID is given, the state of all routines is returned.

Returns:

TRUE if successful, **FALSE** otherwise (see p. 7)

BOOL PI_qFRR (int *ID*, const char* *szScanRoutineNames*, const unsigned int* *iResultIdsArray*, char* *szResult*, int *iBufferSize*)

Corresponding command: FRR?

Fast alignment: Gets the results of a fast alignment routine. See the E712T0016 Technical Note for valid result identifiers and possible results. Use the response to PI_qFRH() to get information on the supported result identifiers.

Arguments:

ID ID of controller

szScanRoutineNames The identifier of the routine. If no routine identifier is given, all available results are queried.

iResultIdsArray The identifier of the result. If no result identifier is given, all available results for the given routine are queried.

szResult buffer to receive the string read in from controller, lines are separated by '\n' ("line-feed")

iBufferSize size of szBuffer, must be given to avoid buffer overflow.")

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qFSS (int *ID*, int* *piResult*)

Corresponding command: FSS?

Gets the status of the last scanning procedure that was started.

In order to check whether a scanning procedure is still going on, the motion status of the axes can be queried with PI_IsMoving().

PI_qFSS() gets the status of scanning procedures that are started with the following commands:

PI_AAP(), PI_FIO(), PI_FLM(), PI_FLS(), PI_FSA(), PI_FSC(), PI_FSM()

Arguments:

ID ID of controller

piResult indicates the status of the last scanning procedure that was started.

1: Scanning procedure has been successfully completed

0: Scanning procedure is still going on or has been unsuccessfully completed.

Returns:

TRUE if successful, **FALSE** otherwise (see p. 7)

BOOL PI_qHAR (int *ID*, const char* *szAxes*, BOOL* *pbValueArray*)

Corresponding command: HAR?

Gets whether the hard stops of the axis can be used for reference moves.

Arguments:

ID ID of controller

szAxes string with axes

pbValueArray indicates whether the axis can be referenced using the hard stop (= 1) or not (= 0).

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qHDR (int *ID*, char* *szBuffer*, int *iBufferSize*)

Corresponding command: HDR?

Lists a help string which contains all information available for data recording (record options and trigger options, information about additional parameters and commands regarding data recording).

For more information see "Data Recorder" in the controller User Manual.

Arguments:**ID** ID of controller**szBuffer** buffer to receive the string read in from controller, lines are separated by '\n' ("line-feed")**iBufferSize** size of **szBuffer**, must be given to avoid buffer overflow.**Returns:****TRUE** if no error, **FALSE** otherwise (see p. 7)

```

BOOL PI_qHDT (int ID, const int* iDeviceIDsArray, const int* iAxisIDsArray, int*
piValueArray, int iArraySize)

```

Corresponding command: HDT?

Gets the currently assigned lookup table for the given axis of the given HID device.

Arguments:**ID** ID of controller**iDeviceIDsArray** HID devices connected to the controller.**iAxisIDsArray** axes of the HID device(s)**piValueArray** lookup tables assigned to the axes of the HID device(s), see PI_HDT() for available tables**iArraySize** size of *iDeviceIDsArray*, *iAxisIDsArray* and *piValueArray***Returns:****TRUE** if successful, **FALSE** otherwise

```

BOOL PI_qHIA (int ID, const char* szAxes, const int* iFunctionArray, int*
iDeviceIDsArray, int* iAxesIDsArray)

```

Corresponding command: HIA?

Gets the current control configuration for the given motion parameter of the given axis of the controller, i. e. the currently assigned axis of an HID device.

Arguments:**ID** ID of controller**szAxes** string with axes of the controller**iFunctionArray** motion parameters to be queried**iDeviceIDsArray** IDs of the HID devices used for HID control**iAxesIDsArray** IDs of the axes of the HID device(s) used for HID control**Returns:****TRUE** if successful, **FALSE** otherwise

```

BOOL PI_qHIB (int ID, const int* iDeviceIDsArray, const int* iButtonIDsArray, int*
pbValueArray, int iArraySize)

```

Corresponding command: HIB?

Gets the current state of the given button of the given HID device.

Arguments:**ID** ID of controller**iDeviceIDsArray** HID devices connected to the controller**iButtonIDsArray** buttons of the HID device(s)**pbValueArray** array to receive the states of the buttons**iArraySize** size of *iDeviceIDsArray*, *iButtonIDsArray* and *pbValueArray***Returns:****TRUE** if successful, **FALSE** otherwise

BOOL PI_qHIE (int *ID*, const int* *iDeviceIDsArray*, const int* *iAxesIDsArray*, double* *pdValueArray*, int *iArraySize*)

Corresponding command: HIE?

Gets the current displacement of the given axis of the given HID device.

Arguments:

ID ID of controller

iDeviceIDsArray HID devices connected to the controller

iAxesIDsArray axes of the HID device(s)

pdValueArray array to receive the displacement of the axes of the HID device(s)

iArraySize size of *iDeviceIDsArray*, *iAxesIDsArray* and *pdValueArray*

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_qHIL (int *ID*, const int* *iDeviceIDsArray*, const int* *iLED_IDsArray*, int* *pnValueArray*, int *iArraySize*)

Corresponding command: HIL?

Gets the current state of the given output unit or characteristic ("LED") of the given HID device.

Arguments:

ID ID of controller

iDeviceIDsArray HID devices connected to the controller

iLED_IDsArray output units or characteristics ("LEDs") of the HID device(s)

pnValueArray array to receive the states of the LEDs

iArraySize size of *iDeviceIDsArray*, *iLED_IDsArray* and *pnValueArray*

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_qHIN (int *ID*, const char* *szAxes*, BOOL* *pbValueArray*)

Corresponding command: HIN?

Gets the activation state of the control by HID devices ("HID control") for the given axis of the controller.

Arguments:

ID ID of controller

szAxes string with axes

pbValueArray array to receive the activation state of the HID control

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_qHIS (int *ID*, char* *szBuffer*, int *iBufferSize*)

Corresponding command: HIS?

Gets the properties of the operating elements of HID devices connected to the controller.

Arguments:

ID ID of controller

szBuffer buffer to receive the string read in from controller, lines are separated by '\n' ("line-feed")

iBufferSize size of *szBuffer*, must be given to avoid buffer overflow.

Returns:

TRUE if no error, FALSE otherwise

```

BOOL PI_qHIT (int ID, const int* piTableIdsArray, int iNumberOfTables, int
iOffsetOfFirstPointInTable, int iNumberOfValues, double** pdValueArray, char*
szGcsArrayHeader, int iGcsArrayHeaderMaxSize)

```

Corresponding command: HIT?

Gets the values of the given points in the given lookup table.

Arguments:

ID ID of controller

piTableIdsArray IDs of the lookup tables of the controller

iNumberOfTables number of tables to read

iOffsetOfFirstPointInTable index of first point to be read (starts with index 1)

iNumberOfValues number of points to read

pdValueArray pointer to internal array to store the data; data from all tables read will be placed in the same array with the values interspersed; the DLL will allocate enough memory to store all data, call `PI_GetAsyncBufferIndex()` to find out how many data points have already been transferred

szGcsArrayHeader buffer to store the GCS array header

iGcsArrayHeaderMaxSize size of the buffer to store the GCS array header, must be given to prevent buffer overflow

Returns:

TRUE if successful, **FALSE** otherwise

```

BOOL PI_qHLP (int ID, char* szBuffer, int iBufferSize)

```

Corresponding command: HLP?

Read in the help string from the controller. The answer is quite long (up to 3000 characters) so be sure to provide enough space! (And you may have to wait a bit...)

Arguments:

ID ID of controller

szBuffer buffer to receive the string read in from controller, lines are separated by '\n' ("line-feed")

iBufferSize size of *szBuffer*, must be given to avoid buffer overflow.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

```

BOOL PI_qHPA (int ID, char* szBuffer, int iBufferSize)

```

Corresponding command: HPA?

Lists a help string which contains all available parameters with short descriptions. See the user manual of the controller for an appropriate list of all parameters available for your controller.

Arguments:

ID ID of controller

szBuffer buffer to receive the string read in from controller, lines are separated by '\n' ("line-feed")

iBufferSize size of *szBuffer*, must be given to avoid buffer overflow.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

```

BOOL PI_qHPV (int ID, char* szBuffer, int iBufferSize)

```

Corresponding command: HPV?

Responds with a help string which contains possible parameters values. Use `PI_qHPA` instead to get a help string which contains all available parameters with short descriptions.

Arguments:

ID ID of controller

szBuffer buffer to receive the string read in from controller, lines are separated by '\n' ("line-feed").

iBufferSize size of *szBuffer*, must be given to avoid buffer overflow.

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_qIDN (int *ID*, char* *szBuffer*, int *iBufferSize*)

Corresponding command: *IDN?

Get identification string of the controller.

Arguments:

ID ID of controller

szBuffer buffer to receive the string read in from controller

iBufferSize size of *szBuffer*, must be given to avoid a buffer overflow.

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_qIFC (int *ID*, const char* *szParameters*, char* *szBuffer*, int *iBufferSize*)

Corresponding command: IFC?

Get the interface configuration from volatile memory.

Arguments:

ID ID of controller

szParameters the interface parameter to be queried, can be RSBAUD, GPADR, IPADR, IPSTART, IPMASK and MACADR (depends on the controller)

szBuffer Buffer to receive the values of the parameters from volatile memory

iBufferSize the size of the buffer *szBuffer*

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_qIFS (int *ID*, const char* *szParameters*, char* *szBuffer*, int *iBufferSize*)

Corresponding command: IFS?

Get the interface configuration stored in non-volatile memory (this is the current power-on default).

Arguments:

ID ID of controller

szParameters the interface parameters to be queried, can be RSBAUD, GPADR, IPADR, IPSTART, IPMASK and MACADR (depends on the controller)

szBuffer buffer to receive the values of the parameters from non-volatile memory

iBufferSize size of *szBuffer*, must be given to avoid a buffer overflow.

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_qIMP (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: IMP?

Get last sent impulse parameters for given axis (sent with PI_IMP()).

Arguments:

ID ID of controller

szAxes axis for which the impulse parameters are to be read

pdValueArray Array to be filled with impulse parameters of the axes; currently only the pulse height.

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

```

BOOL PI_qJAS (int ID, const int* iJoystickIDsArray, const int* iAxesIDsArray,
double* pdValarray, int iArraySize)

```

Corresponding command: JAS?

Get the current status of the given axis of the given joystick device which is directly connected to the controller. The reported factor is applied to the velocity set with PI_VEL() (closed-loop operation) or PI_OVL() (open-loop operation), the range is -1.0 to 1.0.

See "Joystick Control" in the controller User Manual for details.

Arguments:

ID ID of controller

iJoystickIDsArray array with joystick devices connected to the controller

iAxesDsArray array with joystick axes

pdValarray pointer to array to receive the joystick axis amplitude, i.e. the factor which is currently applied to the current valid velocity setting of the controlled motion axis; corresponds to the current displacement of the joystick axis.

iArraySize size of arrays

Returns:

TRUE if successful, FALSE otherwise

```

BOOL PI_qJAX (int ID, const int* iJoystickIDsArray, const int* iAxesIDsArray, int
iArraySize, char* szAxesBuffer, int iBufferSize)

```

Corresponding command: JAX?

Get axis controlled by a joystick axis of a joystick device which is directly connected to the controller.

See "Joystick Control" in the controller User Manual for details.

Arguments:

ID ID of controller

iJoystickIDsArray array with joystick devices connected to the controller

iAxesIDsArray array with IDs of the joystick axes

iArraySize size of arrays

buffer buffer to receive the string read in from controller; will contain axis IDs of axes associated with corresponding joystick axis

maxlen size of *buffer*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, FALSE otherwise

```

BOOL PI_qJBS (int ID, const int* iJoystickIDsArray, const int* iButtonIDsArray,
BOOL* pbValarray, int iArraySize)

```

Corresponding command: JBS?

Get the current status of the given button of the given joystick device which is directly connected to the controller.

See "Joystick Control" in the controller User Manual for details.

Arguments:

ID ID of controller

iJoystickIDsArray array with joystick devices connected to the controller

iButtonIDsArray array with joystick buttons

pbValarray pointer to array to receive the joystick button state, indicates if the joystick button is pressed; 0 = not pressed, 1 = pressed

iArraySize size of arrays

Returns:

TRUE if successful, FALSE otherwise

```

BOOL PI_qJLT (int ID, const int* iJoystickIDsArray, const int* iAxisIDsArray, int
iNumberOfTables, int iOffsetOfFirstPointInTable, int iNumberOfValues, double**
pdValueArray, char* szGcsArrayHeader, int iGcsArrayHeaderMaxSize)

```

Corresponding command: JLT?

Get joystick lookup table values.

Arguments:

ID ID of controller

szAxes string with axes

iJoystickIDsArray array with joystick devices connected to the controller

iAxisIDsArray array with joystick axes

iNumberOfTables number of tables to read

iOffsetOfFirstPointInTable index of first point to be read (starts with index 1)

iNumberOfValues number of points to read

pdValueArray pointer to internal array to store the data; data from all tables read will be placed in the same array with the values interspersed; the DLL will allocate enough memory to store all data, call `PI_GetAsyncBufferIndex()` to find out how many data points have already been transferred

szGcsArrayHeader buffer to store the GCS array header

iGcsArrayHeaderMaxSize size of the buffer to store the GCS array header, must be given to prevent buffer overflow

Returns:

TRUE if successful, **FALSE** otherwise

```

BOOL PI_qJON (int ID, const int* iJoystickIDsArray, BOOL* pbValarray, int
iArraySize)

```

Corresponding command: JON?

Get activation state of the given joystick device which is directly connected to the controller.

See "Joystick Control" in the controller User Manual for details.

Arguments:

ID ID of controller

iJoystickIDsArray array with joystick devices connected to the controller

pbValarray pointer to array to receive the joystick enable states (0 for deactivated, 1 for activated)

iArraySize size of arrays

Returns:

TRUE if successful, **FALSE** otherwise

```

BOOL PI_qKEN (int ID, const char* szNamesOfCoordSystems, char* buffer, int
bufsize)

```

Corresponding command: KEN?

Lists enabled coordinate systems by name.

Arguments:

ID ID of controller

szNamesOfCoordSystems string with name of the coordinate system. Can be NULL or "" to return all

szBuffer buffer to receive the string read in from controller, lines are separated by line-feed characters

iBufferSize size of *szBuffer*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qKET (int *ID*, const char* *szNamesOfCoordSystems*, char* *buffer*, int *bufsize*)

Corresponding command: KET?

Lists enabled coordinate systems by type.

Arguments:

ID ID of controller

szNamesOfCoordSystems string with name of the coordinate system. Can be NULL or "" to return all

szBuffer buffer to receive the string read in from controller, lines are separated by line-feed characters

iBufferSize size of *szBuffer*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_qKLC (int *ID*, const char* *szNamesOfCoordSystem1*, const char* *szNamesOfCoordSystem2*, const char* *szItem1*, const char* *szItem2*, char* *buffer*, int *bufsize*)

Corresponding command: KLC?

Lists properties of combinations of Work and Tool coordinate systems.

Arguments:

ID ID of controller

szNamesOfCoordSystem1 string with name of the coordinate system. Can be NULL or ""

szNamesOfCoordSystem2 string with name of the coordinate system. Can be NULL or ""

szItem1 string with first item to query. Can be NULL or ""

szItem2 string with second item to query. Can be NULL or ""

Buffer buffer to receive the string read in from controller, lines are separated by line-feed characters

iBufferSize size of *szBuffer*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_qKLN (int *ID*, const char* *szNamesOfCoordSystems*, char* *buffer*, int *bufsize*)

Corresponding command: KLN?

Lists coordinate system chains.

Arguments:

ID ID of controller

szNamesOfCoordSystems string with name of the coordinate system. Can be NULL or "" to return all

szBuffer buffer to receive the string read in from controller, lines are separated by line-feed characters

iBufferSize size of *szBuffer*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_qKLS (int *ID*, const char* *szNamesOfCoordSystem*, const char* *szItem1*, const char* *szItem2*, char* *buffer*, int *bufsize*)

Corresponding command: KLS?

Lists properties of all coordinate systems..

Arguments:

ID ID of controller

szNamesOfCoordSystem string with name of the coordinate system. Can be NULL or ""

szItem1 string with first item to query. Can be NULL or ""

szItem2 string with second item to query. Can be NULL or ""

szBuffer buffer to receive the string read in from controller, lines are separated by line-feed characters

iBufferSize size of **szBuffer**, must be given to avoid buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qKLT (int *ID*, const char* *szStartCoordSystem*, const char* *szEndCoordSystem*, char* *buffer*, int *bufsize*)

Corresponding command: KLT?

Returns the position and orientation of the coordinate system which results from a chain of linked coordinate systems, or from a part of a chain. The part to be queried can be limited by specifying the start and end points in the chain.

Arguments:

ID ID of controller

szStartCoordSystem name of the coordinate system which is the start point in the chain. Can be NULL or ""

szEndCoordSystem name of the coordinate system which is the end point in the chain. Can be NULL or ""

szBuffer buffer to receive the string read in from controller, lines are separated by line-feed characters

iBufferSize size of **szBuffer**, must be given to avoid buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qLIM (int *ID*, const char* *szAxes*, BOOL* *pbValueArray*)

Corresponding command: LIM?

Check if the given axes have limit switches.

Arguments:

ID ID of controller

szAxes string with axes, if "" or NULL all axes are queried.

pbValueArray array for limit switch info: **TRUE** if axis has limit switches, **FALSE** if not

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qMAC (int *ID*, const char* *szMacroName*, char* *szBuffer*, int *iBufferSize*)

Corresponding command: MAC?

Get available macros, or list contents of a specific macro. If *szMacroName* is empty or NULL, all available macros are listed in *szBuffer*, separated with line-feed characters. Otherwise the content of the macro with name *szMacroName* is listed, the single lines separated by line-feed characters. If there are no macros stored or the requested macro is empty the answer will be "".

See "Controller Macros" and the MAC command description in the controller User Manual for details.

Arguments:

ID ID of controller

szMacroName string with name of the macro to list

szBuffer buffer to receive the string read in from controller, lines are separated by line-feed characters

iBufferSize size of **szBuffer**, must be given to avoid buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qMAN (int *ID*, const char* *szCommand*, char* *szBuffer*, int *iBufferSize*)

Corresponding command: MAN?

Shows a detailed help text for individual commands.

Arguments:

ID ID of controller

szCommand is the command mnemonic of the command for which the help text is to be displayed.

szBuffer buffer to receive the string that describes the command.

iBufferSize size of *szBuffer*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_qMOD (int *ID*, const char* *szItems*, const unsigned int * *iModeArray*, char* *szValues*, int *iMaxValueSize*)

Corresponding command: MOD?

Get modes for axes / channels / system.

Arguments:

ID ID of controller

szItems string with item identifiers

iModeArray array with IDs of modes to be queried

szValues string to be filled with values for each mode

iMaxValueSize size of *szValues*, must be given to avoid buffer overflow

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_qMOV (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: MOV?

Read the commanded target positions for *szAxes*. Use PI_qPOS() to get the current positions.

Arguments:

ID ID of controller

szAxes string with axes, if "" or NULL all axes are queried.

pdValueArray array to be filled with target positions of the axes

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_qNAV (int *ID*, const int* *piAnalogChannelIds*, int* *piNrReadingsValues*, int *iArraySize*)

Corresponding command: NAV?

Gets the number of readout values of the analog input used for averaging.

The response consists of a line feed when the controller does not contain an analog input channel.

Arguments:

ID ID of controller

piAnalogChannelIds array of analog input channel identifiers

piNrReadingsValues array to be filled with number of readout values used for averaging

iArraySize size of *piAnalogChannelIds* and *piNrReadingsValues*

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_qNLM (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: NLM?

Get lower limits ("soft limits") for the positions of *szAxes*.

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray array to be filled with lower limits for position of the axes.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qOAC (int *ID*, const int* *piPIEZOWALKChannelsArray*, double* *pdValueArray*, int *iArraySize*);

Corresponding command: OAC?

Get current open-loop acceleration of the PiezoWalk channels.

Arguments:

ID ID of controller

piPIEZOWALKChannelsArray array with PiezoWalk channels, if **NULL** all PiezoWalk channels are queried

pdValueArray array to receive the acceleration value

iArraySize size of the arrays *piPIEZOWALKChannelsArray* (if not **NULL**) and *pdValueArray*

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qOAD (int *ID*, const int* *piPIEZOWALKChannelsArray*, double* *pdValueArray*, int *iArraySize*)

Corresponding command: OAD?

Reads last commanded open-loop analog driving voltage of given PiezoWalk channel.

Arguments:

ID ID of controller

piPIEZOWALKChannelsArray array with PiezoWalk channels, if **NULL** all PiezoWalk channels are queried.

pdValueArray array to receive the last-commanded feed voltage amplitude in V

iArraySize size of the arrays *piPIEZOWALKChannelsArray* (if not **NULL**) and *pdValueArray*

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qODC (int *ID*, const int* *piPIEZOWALKChannelsArray*, double* *pdValueArray*, int *iArraySize*);

Corresponding command: ODC?

Get current open-loop deceleration of the PiezoWalk channels.

Arguments:

ID ID of controller

piPIEZOWALKChannelsArray array with PiezoWalk channels, if **NULL** all PiezoWalk channels are queried

pdValueArray array to receive the acceleration value

iArraySize size of the arrays *piPIEZOWALKChannelsArray* (if not **NULL**) and *pdValueArray*

Returns:

TRUE if successful, **FALSE** otherwise

```
BOOL PI_qOMA (int ID, const char* szAxes, double* pdValueArray);
```

Corresponding command: OMA?

Reads last commanded open-loop target *pdValueArray* of given *szAxes*.

Arguments:

ID ID of controller

szAxes string with axes, if "" or **NULL** all axes are queried.

pdValueArray array to be filled with target positions of the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

```
BOOL PI_qONL (int ID, const int* iPiezoChannels, int* pdValarray, int iArraySize)
```

Corresponding command: ONL?

Gets current control mode for *iPiezoChannels*.

Arguments:

ID ID of controller

iPiezoChannels string with piezo channels, if "" or **NULL** all piezo channels are queried

pdValueArray array to receive the control modes of the specified piezo channels, **TRUE** for "ONLINE", **FALSE** for "OFFLINE"

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

```
BOOL PI_qONT (int ID, const char* szAxes, BOOL* pbValueArray)
```

Corresponding command: ONT?

Check if *szAxes* have reached the target.

Arguments:

ID ID of controller

szAxes string with axes

pbValueArray array to be filled with current on-target state of the axes

Returns:

TRUE if successful, **FALSE** otherwise

```
BOOL PI_qOSN (int ID, const int* piPiezoWalkChannelsArray, double* pdValueArray, int iArraySize)
```

Corresponding command: OSN?

Reads the number of steps still to be performed for the given PiezoWalk channel after the last call of PI_OSM().

Arguments:

ID ID of controller

piPiezoWalkChannelsArray array with PiezoWalk channels, if **NULL** all PiezoWalk channels are queried.

pdValueArray array to receive the number of steps which are still to be performed for open-loop step moving of the given PiezoWalk channels

iArraySize size of the arrays *piPiezoWalkChannelsArray* (if not **NULL**) and *pdValueArray*

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

```
BOOL PI_qOVF (int ID, const char* szAxes, BOOL* pbOverflow)
```

Corresponding command: OVF?

Checks overflow status of *szAxes*. Overflow means that the control variables are out of range (can only happen if controller is in closed-loop mode).

Arguments:**ID** ID of controller**szAxes** string with axes**pbOverflow** array to be filled with current overflow status of the axes ("0" = axis is not in overflow or "1" = axis is in overflow)**Returns:****TRUE** if successful, **FALSE** otherwise

BOOL PI_qOVL (int *ID*, const int* *piPIEZOWALKChannelsArray*, double* *pdValueArray*, int *iArraySize*)

Corresponding command: OVL?

Get the current value of the velocity for open-loop nanostepping motion.

Arguments:**ID** ID of controller**piPIEZOWALKChannelsArray** array with PiezoWalk channel identifiers**pdValueArray** array to be filled with the current active velocity values for open-loop nanostepping motion, in steps/s**iArraySize** size of arrays**Returns:****TRUE** if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qPLM (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: PLM?Get upper limits ("soft limit") for the positions of *szAxes*.**Arguments:****ID** ID of controller**szAxes** string with axes**pdValueArray** array to be filled with upper limits for position of the axes.**Returns:****TRUE** if successful, **FALSE** otherwise

BOOL PI_qPOS (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: POS?Get the current positions of *szAxes*. If no position sensor is present in your system, the response to *PI_qPOS()* is not meaningful.To request the current position of input signal channels (sensors) in physical units, use *PI_qTSP()* instead.**Arguments:****ID** ID of controller**szAxes** string with axes, if "" or **NULL** all axes are queried.**pdValueArray** array to receive the current positions of the axes**Returns:****TRUE** if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qPUN (int *ID*, const char* *szAxes*, char* *szUnit*, int *iBufferSize*)

Corresponding command: PUN?Get the position units of *szAxes*.**Arguments:****ID** ID of controller**szAxes** string with axes, if "" or **NULL** all axes are queried.**pdValueArray** array to receive the position units of the axes**Returns:**

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_qRMC (int *ID*, char * *szBuffer*, int *iBufferSize*)

Corresponding command: RMC?

List macros which are currently running.

See "Controller Macros" and the MAC command description in the controller User Manual for details.

Arguments:

ID ID of controller

szBuffer buffer to receive the string read in from controller, lines are separated by line-feed characters. Contains the names of the macros which are saved on the controller and currently running. The response is an empty line when no macro is running.

iBufferSize size of *buffer*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_qRON (int *ID*, const char* *szAxes*, BOOL* *pbValueArray*)

Corresponding command: RON?

Gets reference mode for given axes.

Arguments:

ID ID of controller

szAxes string with axes

pbValarray array to receive reference modes for the specified axes

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_qRTO (int *ID*, const char* *szAxes*, int* *pbValueArray*)

Corresponding command: RTO?

Read the "ready-for-turn-off state" of the given axis (check whether PI_RTO() was successful).

Arguments:

ID ID of controller

szAxes string with axes, if "" or NULL all axes are affected.

pbValueArray array to receive, 1 if ready (i.e. PI_RTO() was successful), 0 if not ready (i.e. PI_RTO() was not successful)

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_qRTR (int *ID*, int* *piRecordTableRate*)

Corresponding command: RTR?

Gets the current record table rate, i.e. the number of servo-loop cycles used in data recording operations.

Arguments:

ID ID of controller

piRecordTableRate variable to be filled with the record table rate

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_qSAI (int *ID*, char* *szAxes*, int *iBufferSize*)

Corresponding command: SAI?

Get the identifiers for all configured axes. Each character in the returned string is an axis identifier for one logical axis.

Deactivated axes are not shown.

Arguments:

ID ID of controller

szAxes buffer to receive the string read in

iBufferSize size of *szAxes*, must be given to avoid a buffer overflow.

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_qSAI_ALL (int *ID*, char* *szAxes*, int *iBufferSize*)

Corresponding command: SAI?

Get the identifiers for all axes (configured and unconfigured axes). Each character in the returned string is an axis identifier for one logical axis. This function is provided for compatibility with controllers which allow for axis deactivation. PI_qSAI_ALL() then ensures that the answer also includes the axes which are "deactivated".

Arguments:

ID ID of controller

szAxes buffer to receive the string read in

iBufferSize size of *szAxes*, must be given to avoid a buffer overflow.

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_qSCT (int *ID*, double* *pdCycleTime*)

Corresponding command: SCT?

Gets the current cycle time for running a defined motion profile.

Arguments:

ID ID of controller

pdCycleTime cycle time in ms

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_qSEP (int *ID*, const char* *szAxes*, const int* *piParameterArray*, double* *pdValueArray*, char* *szStrings*, int *iMaximumStringSize*)

Corresponding command: SEP?

Query specified parameters for *szAxes* from non-volatile memory. For each desired parameter you must specify a designator in *szAxes* and the parameter ID in the corresponding element of *iParameterArray*.

See the user manual of the controller for a list of the available parameters.

Arguments:

ID ID of controller

szAxes string with designator, one parameter is read for each designatorID in *szAxes*

piParameterArray parameter IDs

pdValueArray array to receive the values of the requested parameters

szStrings string to receive the with linefeed-separated parameter values; when not needed set to NULL (i.e. if numeric parameter values are queried)

iMaximumStringSize size of *szStrings*, must be given to avoid a buffer overflow.

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_qSGA (int *ID*, const int* *piAnalogChannelIds*, int* *piGainValues*, int *iArraySize*)

Corresponding command: SGA?

Gets the gain value *piGainValues* for the given analog input channel *piAnalogChannelIds*.

The response consists of a line feed when the controller does not contain an optical analog input channel.

Arguments:

ID ID of controller

piAnalogChannelIds identifier of the analog input channel

piGainValues array to be filled with gain factor values

iArraySize size of *piAnalogChannelIds* and *piGainValues*

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_qSIC (int *ID*, const int* *piFastAlignmentInputIdsArray*, int *iNumberOfInputIds*, char* *szBuffer*, int *iBufferSize*)

Corresponding command: SIC?

Fast alignment: Gets the calculation settings for the given fast alignment input channel.

The calculation results can be queried with PI_qTCI().

See the E712T0016 Technical Note ("Fast Alignment Routines") for detailed descriptions of the fast alignment input channels.

Arguments:

ID ID of controller

piFastAlignmentInputIdsArray The identifier of a fast alignment input channel of the controller.

iNumberOfInputIds size of *piFastAlignmentInputIdsArray*

szBuffer buffer to receive the string read in from controller, lines are separated by '\n' ("line-feed").

iBufferSize size of *szBuffer*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_qSMO (int *ID*, char *const *szAxes*, int * *pnValueArray*)

Corresponding command: SMO?

Gets last valid control value of *szAxes*.

Arguments:

ID ID of controller

szAxes string with axes, if "" or NULL all axes are affected.

pnValueArray control values for the specified axes. In servo-on mode the current value, set by the controller, is reported. In servo-off mode the value set by PI_SMO() is reported.

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_qSPA (int *ID*, const char* *szAxes*, unsigned int* *piParameterArray*, double* *pdValueArray*, char* *szStrings*, int *iMaxNameSize*)

Corresponding command: SPA?

Query specified parameters for *szAxes* from RAM. For each desired parameter you must specify a designator in *szAxes* and the parameter ID in the corresponding element of *iParameterArray*. See the user manual of the controller for a list of the available parameters.

Arguments:

ID ID of controller

szAxes string with designator, one parameter is read for each designator in *szAxes*

piParameterArray parameter IDs

pdValueArray array to be filled with the values of the requested parameters

szStrings string to receive the linefeed-separated parameter values; when not needed set to **NULL** (i.e. if numeric parameter values are queried)
iMaxNameSize size of **szStrings**, must be given to avoid a buffer overflow.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qSPI (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: SPI?

Gets the pivot point coordinates in the volatile memory.

Arguments:

ID ID of controller

szAxes can be R, S and T. X, Y and Z can also be used as alias identifiers for R, S and T

pdValueArray value array of the pivot point coordinates in physical units

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qSRG (int *ID*, char *const *szAxes*, int * *iRegisterarray*, int * *iValarray*)

Corresponding command: SRG?

Returns register values for queried axes and register numbers.

Arguments:

ID ID of controller

szAxes axis for which the register values should be read

iRegisterarray IDs of registers

iValarray array to be filled with the values for the registers. The answer is bit-mapped and returned as the sum of the individual codes, in hexadecimal format.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qSSA (int *ID*, const int* *iPIEZOWALKChannels*, double* *pdValueArray*, int *iArraySize*)

Corresponding command: SSA?

Get the current value of the voltage amplitude used for nanostepping motion.

Arguments:

ID ID of controller

piPIEZOWALKChannelsArray array with PiezoWalk channels

pdValueArray array to be filled with the current active voltage amplitude values in V

iArraySize size of the arrays **piPIEZOWALKChannelsArray** and **pdValueArray**

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qSSL (int *ID*, const char* *szAxes*, BOOL* *pbValueArray*)

Corresponding command: SSL?

Gets the state of the soft limits that are set with PI_NLM() and PI_PLM(). If all arguments are omitted, the state is queried for all axes.

Arguments:

ID ID of controller

szAxes axes of the controller

pbValueArray array to receive the state of the soft limits:

0 = soft limits deactivated

1 = soft limits activated

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qSSN (int *ID*, char* *szSerialNumber*, int *iBufferSize*)

Corresponding command: SSN?

Get serial number of the controller.

Arguments:

ID ID of controller

szSerialNumber buffer for storing the string read in

iBufferSize size of buffer, must be given to avoid buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qSST (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: SST?

Gets the distance ("step size") for motions of the given axis that are triggered by a manual control unit.

Arguments:

ID ID of controller

szAxes axes of the controller, if "" or **NULL** all axes are queried.

pdValueArray array to receive the distance values used for the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qSTE (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: STE?

Get last sent amplitude for the step response measurement for given axis (sent with PI_STE()).

Arguments:

ID ID of controller

szAxes axes to be read

pdValueArray array to be filled with the step amplitude values of the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qSVA (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: SVA?

Returns last valid open-loop control value for *szAxes*.

Arguments:

ID ID of controller

szAxes string with axes, if "" or **NULL** all axes are queried

pdValueArray array to be filled with the last commanded open-loop control values (dimensionless).

The interpretation of the open-loop control values depends on the controller.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qSVO (int *ID*, const char* *szAxes*, BOOL* *pbValueArray*)

Corresponding command: SVO?

Get the servo-control mode for *szAxes*

Arguments:

ID ID of controller

szAxes string with axes, if "" or **NULL** all axes are queried

pbValueArray array to receive the servo modes of the specified axes, **TRUE** for "on", **FALSE** for "off"

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qTAC (int *ID*, int* *piNrChannels*)

Corresponding command: TAC?

Get the number of installed analog channels.

Arguments:

ID ID of controller

piNrChannels pointer to int to receive the number of installed analog channels

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qTAD (int *ID*, const int* *piSensorChannelsArray*, int* *piValueArray*, int *iArraySize*)

Corresponding command: TAD?

Returns ADC value for the given input signal channel, without filtering, linearization and transformation.

The response consists of a line feed when the controller does not contain an analog input channel.

Arguments:

ID ID of controller

piSensorChannelsArray array with input signal channels, if **NULL** all channels are queried.

pdValueArray array to receive ADC value (dimensionless)

iArraySize size of the arrays *pdValueArray* and *piSensorChannelsArray* (if not **NULL**)

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qTAV (int *ID*, const int* *piChannelsArray*, double* *pdValueArray*, int *iArraySize*)

Corresponding command: TAV?

Returns voltage value for the specified analog input channel.

The response consists of a line feed when the controller does not contain an analog input channel.

Arguments:

ID ID of controller

piChannelsArray string with channels. If "" or **NULL** all analog input channels are queried.

pdValueArray array to receive voltage value (in volts)

iArraySize size of *pdValueArray*

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qTCI (int *ID*, const int* *piFastAlignmentInputIDsArray*, double* *pdCalculatedInputValueArray*, int *iArraySize*)

Corresponding command: TCI?

Fast alignment: Gets calculated value of given fast alignment input channel.

The calculation settings of a fast alignment input channel can be defined with `PI_SIC()` and queried with `PI_qSIC()`.

See the E712T0016 Technical Note ("Fast Alignment Routines") for detailed descriptions of the fast alignment input channels.

Arguments:

ID ID of controller

piFastAlignmentInputIDsArray The identifier of a fast alignment input channel of the controller.

pdCalculatedInputValueArray The current value of the calculated input.

iArraySize size of `pdCalculatedInputValueArray`

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qTCV (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: TCV?

Gets the current value of the velocity for closed-loop operation (value calculated by the profile generator).

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray array to be filled with the current velocity values calculated by the profile generator

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qTIO (int *ID*, int* *piInputNr*, int* *piOutputNr*)

Corresponding command: TIO?

Returns the number of available digital I/O channels.

Arguments:

ID ID of controller

piInputNr variable to receive number of available digital input channels

piOutputNr variable to receive number of available digital output channels

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qTMN (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: TMN?

Get the low end of the travel range of *szAxes*

Arguments:

ID ID of controller

szAxes string with axes, if "" or **NULL** all axes are queried.

pdValueArray array to receive low end of the travel range of the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qTMX (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: TMX?

Get the high end of the travel range of *szAxes*.

Arguments:

ID ID of controller

szAxes string with axes, if "" or **NULL** all axes are queried

pdValueArray array to receive high end of travel range of the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qTNR (int *ID*, int* *piNumberOfRecordTables*)

Corresponding command: TNR?

Returns the number of data recorder tables.

For more information see "Data Recorder" in the controller User Manual.

Arguments:

ID ID of controller

piNumberOfRecordTables variable to receive number of data recorder tables

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qTNS (int *ID*, const int* *piSensorChannelsArray*, double* *pdValueArray*, int *iArraySize*)

Corresponding command: TNS?

Returns normalized value for the specified input signal channel.

Arguments:

ID ID of controller

piSensorChannelsArray array with input signal channels, if **NULL** all channels are queried.

pdValueArray array to receive normalized value (dimensionless)

iArraySize the size of the arrays *pdValueArray* and *piSensorChannelsArray* (if not **NULL**)

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qTPC (int *ID*, int* *piNumberOfPiezoChannels*)

Corresponding command: TPC?

Get the number of output signal channels available on the controller.

Arguments:

ID ID of controller

piNumberOfPiezoChannels variable to receive number of available output signal channels

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qTRA (int *ID*, const char* *szAxes*, const double* *pdComponents*, double* *pdValueArray*)

Corresponding command: TRA?

This command returns the maximum absolute position which can be reached from the current position in the given direction for the queried axis vector.

Arguments:

ID ID of controller

szAxes string with axes

pdComponents components of the vector

pdValueArray array to receive maximum positions of the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qTRI (int *ID*, const int* *piTriggerInputIds*, BOOL* *pbTriggerState*, int *iArraySize*)

Corresponding command: TRI?

Returns if the trigger input configuration made with PI_CTI() is enabled or disabled for the given digital input line.

Arguments:

ID ID of controller

piTriggerInputIds digital input lines of the controller

pbTriggerState the current states of the digital input lines:

FALSE = Trigger input disabled

TRUE = Trigger input enabled

iArraySize size of the arrays

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_qTRO (int *ID*, const int* *piTriggerOutputIds*, BOOL* *pbTriggerState*, int *iArraySize*)

Corresponding command: TRO?

Returns if the trigger output configuration made with PI_CTO() is enabled or disabled for the given digital output line.

Arguments:

ID ID of controller

piTriggerOutputIds digital output lines of the controller

pbTriggerState the current states of the digital output lines:

FALSE = Trigger output disabled

TRUE = Trigger output enabled

iArraySize size of the arrays

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_qTRS (int *ID*, const char* *szAxes*, BOOL* *pbValueArray*)

Corresponding command: TRS?

Ask if *szAxes* have reference sensors with direction sensing.

Arguments:

ID ID of controller

szAxes string with axes, if "" or NULL all axes are queried.

pbValueArray array for reference sensor info: TRUE if axis has a reference sensor with direction sensing, FALSE if not

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_qTSC (int *ID*, int* *piNumberOfSensorChannels*)

Corresponding command: TSC?

Get the number of input signal channels available on the controller.

Arguments:

ID ID of controller

piNumberOfSensorChannels variable to receive number of input signal channels

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_qTSP (int *ID*, const int* *piSensorChannelsArray*, double* *pdValueArray*, int *iArraySize*)

Corresponding command: TSP?

Requests the current position of the given input signal channel in physical units (µm).

Arguments:

ID ID of controller

piSensorChannelsArray array with input signal channels, if **NULL** all channels are queried.

pdValueArray array to receive channel position (in µm)

iArraySize the size of the arrays *pdValueArray* and *piSensorChannelsArray* (if not **NULL**)

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qTVI (int *ID*, char * *szBuffer*, int *iBufferSize*)

Corresponding GCS command: TVI?

Get valid characters for axes. Each character in the returned string is a valid axis identifier that can be used to "name" an axis with PI_SAI().

Arguments:

ID ID of controller

szBuffer buffer to store the read in string

iBuffer size of *szBuffer*, must be given to avoid a buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qVAR (int *ID*, const char * *szVariables*, char* *szValues*, int *iBufferSize*)

Corresponding command: VAR?

Gets variable value.

If PI_qVAR is combined with PI_CPY(), PI_JRC(), PI_MEX() or PI_WAC(), the response to PI_qVAR() has to be a single value and not more.

More information regarding local and global variables can be found in "Variables" in the controller User Manual.

Arguments:

ID ID of controller

szVariables name of the variable to be queried

szValues is the value to which the variable is set

iBufferSize size of *szVariables* and *szValues*, must be given to avoid buffer overflow

Returns:

TRUE if successful, **FALSE** otherwise.

BOOL PI_qVCO (int *ID*, char* *szAxes*, BOOL* *pbValueArray*)

Corresponding command: VCO?

Get the velocity-control mode for *szAxes*

Arguments:

ID ID of controller

szAxes string with axes

pbValueArray array to be filled with the velocity-control modes of the specified axes, **TRUE** for "on", **FALSE** for "off"

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qVEL (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: VEL?

Gets the velocity value commanded with PI_VEL() for *szAxes*.

Arguments:

ID ID of controller

szAxes string with axes, if "" or **NULL** all axes are queried.

pdValueArray array to be filled with the velocity settings of the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qVER (int *ID*, char* *szVersion*, int *iBufferSize*)

Corresponding command: VER?

Reports the versions of the controller firmware and the underlying drivers and libraries.

Arguments:

ID ID of controller

szVersion buffer for storing the string read in

iBufferSize size of *szVersion*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qVLS (int *ID*, double* *pdSystemVelocity*)

Corresponding command: VLS?

Gets the velocity of the moving platform of the Hexapod that is set with PI_VLS().

Arguments:

ID ID of controller

pdSystemVelocity velocity value in physical units

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qVMA (int *ID*, const int* *piPiezoChannelsArray*, double* *pdValueArray*, int *iArraySize*)

Corresponding command: VMA?

Get upper piezo voltage soft limit for *piPiezoChannelsArray*.

Arguments:

ID ID of controller

piPiezoChannelsArray array with piezo channels, if **NULL** all piezo channels are queried.

pdValueArray array to be filled with the upper limits for the piezo voltage

iArraySize size of the arrays *pdValueArray* and *piPiezoChannelsArray* (if not **NULL**)

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qVMI (int *ID*, const int* *piPiezoChannelsArray*, double* *pdValueArray*, int *iArraySize*)

Corresponding command: VMI?

Get lower piezo voltage soft limit for *piPiezoChannelsArray*.

Arguments:

ID ID of controller

piPiezoChannelsArray array with piezo channels, if **NULL** all piezo channels are queried

pdValueArray array to be filled with the lower limits for the piezo voltage

iArraySize size of the arrays *pdValueArray* and *piPiezoChannelsArray* (if not **NULL**)

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qVMO (int *ID*, const char* *szAxes*, const double* *pdValarray*, BOOL* *pbMovePossible*)

Corresponding command: VMO?

Checks whether the moving platform of the Hexapod can approach a specified position from the current position.

PI_qVMO() does not trigger any motion.

Arguments:

ID ID of controller

szAxes string with axes

pdValarray array of target positions

pbMovePossible value to receive, indicates whether the moving platform can approach the position resulting from the given target position values:

0 = specified position cannot be approached

1 = specified position can be approached

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qVOL (int *ID*, const int* *piPiezoChannelsArray*, double* *pdValueArray*, int *iArraySize*)

Corresponding command: VOL?

Get current piezo voltages for *piPiezoChannelsArray*.

Arguments:

ID ID of controller

piPiezoChannelsArray array with channels, if **NULL** all channels are queried

pdValueArray array to be filled with the current voltages for the channels

iArraySize size of the arrays *pdValueArray* and *piPiezoChannelsArray* (if not **NULL**)

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qVST (int *ID*, char * *szBuffer*, int *iBufferSize*)

Corresponding command: VST?

Get the names of the available stage types. The available stages are read from stage database(s). Depending on the controller, the stage databases are stored on the PC or on the controller.

The stage types listed with PI_qVST() can be assigned to the axes of the controller with PI_CST().

Arguments:

ID ID of controller

szBuffer buffer for storing the string read in, lines are separated by \n (line feed)

iBufferSize size of *szBuffer*, must be given to avoid a buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_RBT (int *ID*)**Corresponding command:** RBT

Reboot Controller. Controller behaves like after a cold start.

Arguments:*ID* ID of controller**Returns:****TRUE** if successful, **FALSE** otherwise**BOOL PI_RNP (int *ID*, const int* *piPIEZOWALKChannelsArray*, const double* *pdValueArray*, int *iArraySize*)****Corresponding command:** RNP

"Relax" the piezos of a given PiezoWalk channel without motion.

Arguments:*ID* ID of controller*piPIEZOWALKChannelsArray* string with PiezoWalk channels*pdValueArray* voltages which must be applied for the PiezoWalk channels, must be 0 to set the voltages to 0 V*iArraySize* size of the arrays *pdValueArray* and *piPIEZOWALKChannelsArray***Returns:****TRUE** if no error, **FALSE** otherwise (see p. 7)**BOOL PI_RON (const int *ID*, char *const *szAxes*, BOOL * *pbValarray*)****Corresponding command:** RON

Sets referencing mode for given axes. Determines how to reference axes measured by incremental sensors.

Arguments:*ID* ID of controller*szAxes* string with axes*pbValarray* reference modes for the specified axes:**Returns:****TRUE** if successful, **FALSE** otherwise**BOOL PI_RPA (int *ID*, const char* *szAxes*, const int* *piParameterArray*)****Corresponding command:** RPACopy specified parameters for *szAxes* from the non-volatile memory and write them to RAM. For each desired parameter you must specify a designator in *szAxes*, and the parameter ID in the corresponding element of *iParameterArray*. See the user manual of the controller for a list of available parameters.**Arguments:***ID* ID of controller*szAxes* string with designators, one parameter is copied for each designator in *szAxes**piParameterArray* parameter IDs**Returns:****TRUE** if no error, **FALSE** otherwise (see p. 7)

BOOL PI_RTO (int *ID*, const char* *szAxes*)**Corresponding command:** RTO

Make ready to turn off: The current position of the given axis is written to the non-volatile memory of the controller. Especially for systems which use incremental sensors. Check with PI_qRTO if PI_RTO() was successful.

Arguments:

ID ID of controller

szAxes string with axes, if "" or **NULL** all axes are affected.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_RTR (int *ID*, int *iRecordTableRate*)**Corresponding command:** RTR

Sets the record table rate, i.e. the number of servo-loop cycles to be used in data recording operations. Settings larger than 1 make it possible to cover longer time periods with a limited number of points.

For more information see "Data Recorder" in the controller User Manual

Arguments:

ID ID of controller

iRecordTableRate is the record table rate to be used (unit: number of servo-loop cycles), must be larger than zero

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_SAI (int *ID*, char *const *szOldAxes*, char *const *szNewAxes*)**Corresponding command:** SAI

Rename axes: *szOldAxes* will be set to *szNewAxes*. The characters in *szNewAxes* must not be in use for any other existing axes and must each be one of the valid identifiers. All characters in *szNewAxes* will be converted to uppercase letters. Only the **last** occurrence of an axis identifier in *szNewAxes* will be used to change the name.

Arguments:

ID ID of controller

szOldAxes old axis identifiers

szNewAxes new identifiers for the axes

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_SCT (int *ID*, double *dCycleTime*)**Corresponding command:** SCT

Determines the cycle time for running a motion profile.

Arguments:

ID ID of controller

dCycleTime cycle time in ms

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_SEP (int *ID*, const char* *szPassword*, const char* *szAxes*, const int* *piParameterArray*, const double* *pdValueArray*, const char* *szStrings*)

Corresponding command: SEP

Set specified parameters for *szAxes* in non-volatile memory. For each parameter you must specify a designator in *szAxes*, and the parameter ID in the corresponding element of *iParameterArray*. See the user manual of the controller for a list of available parameters.

Notes:

If the same designator has the same parameter number more than once, only the **last** value will be set. For example PI_SEP(id, "100", "111", {0x1, 0x1, 0x2}, {3e-2, 2e-2, 2e-4}) will set the P-term of '1' to 2e-2 and the I-term to 2e-4.

After parameters were set with PI_SEP(), use PI_RPA() to activate them (write them to volatile memory), or they become active after next power-on or reboot.

Warnings:

This command is for setting hardware-specific parameters. Wrong values may lead to improper operation or damage of your hardware!

The number of write times of non-volatile memory is limited. Do not write parameter values except when necessary.

Arguments:

ID ID of controller

szPassword There is a password required to set parameters in the non-volatile memory. This password is "100"

szAxes string with designators, one parameter is set for each designator in *szAxes*

piParameterArray Parameter IDs

pdValueArray array with the values for the respective parameters

szStrings string with linefeed-separated parameter values; when not needed set to **NULL** (i.e. if numeric parameter values are used)

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_SGA (int *ID*, const int* *piAnalogChannelIds*, const int* *piGainValues*, int *iArraySize*)

Corresponding command: SGA

Determines the gain value for the given analog input channel.

Arguments:

ID ID of controller

piAnalogChannelIds array of analog input channel identifiers

piGainValues array of gain factors

iArraySize size of *piAnalogChannelIds* and *piGainValues*

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_SIC (int *ID*, int *iFastAlignmentInputId*, int *iCalcType*, const double* *pdParameters*, int *iNumberOfParameters*)

Corresponding command: SIC

Fast alignment: Defines calculation settings for the given fast alignment input channel.

The current valid calculation settings can be queried with PI_qSIC(). The calculation results can be queried with PI_qTCI().

See the E712T0016 Technical Note ("Fast Alignment Routines") for detailed descriptions.

Arguments:

ID ID of controller

iFastAlignmentInputId The identifier of a fast alignment input channel of the controller.

iCalcType The type of calculation to be applied, can be:

- 0 = No calculation
- 1 = Exponential calculation
- 2 = Polynomial calculation

pdParameters The settings for the selected calculation type.

iNumberOfParameters size of **pdParameters**.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_SMO (int *ID*, char *const *szAxes*, int * *pnValueArray*)

Corresponding command: SMO

Sets control value directly to move the axis. Profile generator (if present), sensor feedback and servo algorithm are not taken into account. This is only possible if servo-control is OFF (see PI_SVO()).

CAUTION: In the case of large control values, the stage can strike the hard stop despite the limit switch function. This can cause damage to equipment.

Arguments:

- ID** ID of controller
- szAxes** string with axes
- pnValueArray** array with control values.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_SPA (int *ID*, const char* *szAxes*, const unsigned int* *piParameterArray*, const double* *pdValueArray*, const char* *szStrings*)

Corresponding command: SPA

Set specified parameters for *szAxes* in RAM. For each parameter you must specify a designator in *szAxes*, and the parameter ID in the corresponding element of *iParameterArray*. See the user manual of the controller for a list of available parameters.

Notes:

If the same designator has the same parameter number more than once, only the **last** value will be set. For example PI_SPA(id, "111", {0x1, 0x1, 0x2}, {3e-2, 2e-2, 2e-4}) will set the P-term of '1' to 2e-2 and the I-term to 2e-4.

Warning:

This command is for setting hardware-specific parameters. Wrong values may lead to improper operation or damage of your hardware!

Arguments:

- ID** ID of controller
- szAxes** string with designators, one parameter is set for each designator in *szAxes*
- piParameterArray** Parameter IDs
- pdValueArray** array to receive with the values for the respective parameters
- szStrings** string, with linefeed-separated parameter values; when not needed set to **NULL** (i.e. if numeric parameter values are used)

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_SPI (int *ID*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: SPI

Sets the pivot point coordinates in the volatile memory.

Arguments:

- ID** ID of controller
- szAxes** can be R, S and T. X, Y and Z can also be used as alias identifiers for R, S and T
- pdValueArray** value array of the pivot point coordinates

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_SSA (int *ID*, const int* *piPIEZOWALKChannelsArray*, const double* *pdValueArray*, int *iArraySize*)

Corresponding command: SSA

Set the voltage amplitude for nanostepping motion of given PiezoWalk channel.

Arguments:

ID ID of controller

piPIEZOWALKChannelsArray string with PiezoWalk channels

pdValueArray the voltage amplitude for nanostepping motion, in V

iArraySize the size of the arrays *piPIEZOWALKChannelsArray* and *pdValueArray*

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_SSL (int *ID*, const char* *szAxes*, const BOOL* *pbValueArray*)

Corresponding command: SSL

Activates or deactivates the soft limits that are set with PI_NLM() and PI_PLM(). Soft limits can only be activated/deactivated when the axis is not moving (query with PI_IsMoving()).

Arguments:

ID ID of controller

szAxes axes of the controller

pbValueArray array with the states of the soft limits:

0 = soft limits deactivated

1 = soft limits activated

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_SST (int *ID*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: SST

Sets the distance ("step size") for motions of the given axis that are triggered by a manual control unit.

Arguments:

ID ID of controller

szAxes axes of the controller

pdValueArray value array of the distance values

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_STE (int *ID*, const char* *szAxes*, const double* *pdStepSize*)

Corresponding command: STE

Starts performing a step and recording the step response for the given axis.

Arguments:

ID ID of controller

szAxes axes for which the step response will be recorded

pdStepSize amplitude of the step

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_StopAll (int *ID*)

Corresponding command: #24

Stops the motion of all axes instantaneously. Sets error code to 10. This function is identical in function to PI_STP(), but only one character is sent via the interface.

Arguments:

ID ID of controller

Returns:

TRUE if successful, **FALSE** otherwise (see p. 7)

BOOL PI_STP (int *ID*)

Corresponding command: STP

Stops the motion of all axes instantaneously. Sets error code to 10.

PI_STP() also stops macros.

After the axes are stopped, their target positions are set to their current positions.

Arguments:

ID ID of controller

Returns:

TRUE if successful, **FALSE** otherwise (see p. 7)

BOOL PI_SVA (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: SVA

Set absolute open-loop control value to move *szAxes*.

Servo must be switched off (open-loop operation) when using this command.

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray absolute open-loop control value

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_SVO (int *ID*, const char* *szAxes*, const BOOL* *pbValueArray*)

Corresponding command: SVO

Set servo-control "on" or "off" (closed-loop/open-loop mode).

Arguments:

ID ID of controller

szAxes string with axes

pbValueArray servo modes for the specified axes, **TRUE** for "on", **FALSE** for "off"

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_SVR (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: SVR

Set open-loop control value relative to the current open-loop control value to move *szAxes*.

The new open-loop control value is calculated by adding the given value to the last commanded open-loop control value.

Servo must be switched off when using this command (open-loop operation).

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray the open-loop control values which are added to the current values

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_TRI (int *ID*, const int* *piTriggerInputIds*, const BOOL* *pbTriggerState*, int *iArraySize*)

Corresponding command: TRI

Enables or disables the trigger input mode which was set with PI_CTI() for the given digital input line.

Arguments:

ID ID of controller

piTriggerInputIds is an array with the digital input lines of the controller.

pbTriggerState pointer to boolean array with modes for the specified trigger lines, **TRUE** for "on", **FALSE** for "off"

iArraySize number of trigger lines

Returns:

TRUE if successful, **FALSE** otherwise (see p. 7)

BOOL PI_TRO (int *ID*, const int* *piTriggerOutputIds*, const BOOL* *pbTriggerState*, int *iArraySize*)

Corresponding command: TRO

Enables or disables the trigger output mode which was set with PI_CTO() for the given digital output line.

Arguments:

ID ID of controller

piTriggerOutputIds is an array with the digital output lines of the controller.

pbTriggerState pointer to boolean array with modes for the specified trigger lines, **TRUE** for "on", **FALSE** for "off"

iArraySize number of trigger lines

Returns:

TRUE if successful, **FALSE** otherwise (see p. 7)

BOOL PI_VAR (int *ID*, const char * *szVariable*, const char* *szValue*)

Corresponding command: VAR

Sets a variable to a certain value.

Local variables can be set using PI_VAR() in macros only.

The variable is present in RAM only.

See "Variables" and "Controller Macros" in the controller User Manual for details.

Arguments:

ID ID of controller

szVariable name of the variable whose value is to be set

szValue is the value to which the variable is to be set. If omitted, the variable is deleted.

Returns:

TRUE if successful, **FALSE** otherwise.

BOOL PI_VCO (int *ID*, char* *szAxes*, BOOL* *pbValueArray*)

Corresponding command: VCO

Set velocity-control "on" or "off". When velocity-control is "on", the corresponding axes will move with the currently valid velocity. That velocity can be set with PI_VEL() (p. 103).

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray modes for the specified axes, **TRUE** for "on", **FALSE** for "off"

Returns:

TRUE if successful, **FALSE** otherwise (see p. 7)

BOOL PI_VEL (int *ID*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: VEL

Set the velocities to use during moves of *szAxes*. The PI_VEL() setting only takes effect when the given axis is in closed-loop operation (servo on).

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray velocities for the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_VLS (int *ID*, double *dSystemVelocity*)

Corresponding command: VLS

Sets the velocity for the moving platform of the Hexapod.

Arguments:

ID ID of controller

dSystemVelocity velocity value in physical units

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_VMA (int *ID*, const int* *piPiezoChannelsArray*, double* *pdValueArray*, int *iArraySize*)

Corresponding command: VMA

Set upper PZT voltage soft limit of given piezo channel.

Arguments:

ID ID of controller

piPiezoChannelsArray array with piezo channels

pdValueArray upper limits for piezo voltage

iArraySize the size of the arrays *piPiezoChannelsArray* and *pdValueArray*

Returns:

TRUE if successful, **FALSE** otherwise (see p. 7)

BOOL PI_VMI (int *ID*, const int* *piPiezoChannelsArray*, double* *pdValueArray*, int *iArraySize*)

Corresponding command: VMI

Set lower PZT voltage soft limit of given piezo channel.

Arguments:

ID ID of controller

piPiezoChannelsArray array with piezo channels

pdValueArray lower limits for piezo voltage

iArraySize the size of the arrays *piPiezoChannelsArray* and *pdValueArray*

Returns:

TRUE if successful, **FALSE** otherwise (see p. 7)

BOOL PI_WAC (int *ID*, const char * *szCondition*)**Corresponding command:** WAC

Wait until a given condition of the following type occurs: a specified value is compared with a queried value according a specified rule.

Can only be used in macros.

See also PI_MEX()

See "Controller Macros" in the controller User Manual for details.

Valid for:

E-861, C-867, C-887, C-863, C-884, E-871

Arguments:

ID ID of controller

szCondition string with condition to evaluate

Returns:

TRUE if successful, **FALSE** otherwise (see p. 7)

BOOL PI_WPA (int *ID*, const char* *szPassWord*, const char* *szAxes*, const int* *piParameterArray*)**Corresponding command:** WPA

Gets values of the specified parameters from RAM and copies them to non-volatile memory. For each parameter you must specify a designator in *szAxes* and the parameter ID in the corresponding element of *piParameterArray*. See the user manual of the controller for a list of available parameters.

CAUTION: If current parameter values are incorrect, the system may malfunction. Be sure that you have the correct parameter settings before using PI_WPA().

Settings not saved with PI_WPA() will be lost when the controller is powered off or rebooted.

Arguments:

ID ID of controller

szPassWord The password for writing to non-volatile memory depends on the parameter. See the parameter overview and the description of the WPA command in the user manual of the controller.

szAxes string with designators. For each designator in *szAxes* one parameter value is copied.

piParameterArray Array with parameter IDs

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

7. Functions for GCS Commands for Wave Generator and DDL

The functions listed in this chapter are relevant for controllers which support the wave generator functionality and the Dynamic Digital Linearization (DDL).

The listed functions are based on the commands of the GCS. You can use a function only if the corresponding command is supported by your controller. See the user manual of the controller for the supported commands.

For all details regarding the functionality and arguments of commands, see the command descriptions in the user manual of the controller.

NOTE

If a query command is sent as string using `PI_Gcs_Commandset()` it is necessary to make sure that the size of the response string matches the size of the input buffer.

Otherwise it may happen that a response has not yet been retrieved completely before a next function is processed.

See "Functions for Sending and Reading Strings" (p. 17) for details.

Wave Generator

Each axis can be controlled by a "wave generator" which outputs user-specified patterns, so-called "waveforms". This feature is especially important in dynamic applications which require periodic, synchronous motion of the axes. The waveforms to be output are stored in "wave tables" in the controllers volatile memory—one waveform per wave table. Waveforms can be created based on predefined "curve" shapes. Programmable trigger inputs and outputs facilitate synchronization of external events. See "Wave Generator" in the user manual of the controller for more information and for examples.

During the wave generator output, data is recorded in "record tables" on the controller. See "Data Recording" in the controllers User Manual for more information.

The different software interfaces provided for the controller also support use of the wave generator. Waveforms can be defined, stored and displayed in and by the software in a more user-friendly way. If using the wave generator with PIMikroMove, NanoCapture or LabView, read the descriptions in the associated software manual first.

Dynamic Digital Linearization (DDL)

The DDL option can be used in conjunction with the wave generator output in addition to the "normal" servo algorithm in closed-loop operation. The DDL makes it possible to achieve significantly better position accuracy for dynamic applications with periodic motion. DDL "observes" axis motion over one or more wave generator output cycles (DDL initialization). The information gathered is written to "DDL tables" and can then be used to refine the control output signals. The DDL feature must be expressly ordered. You can activate it after purchase and without opening the device. See "Dynamic Digital Linearization (DDL)" in the controllers User Manual for more information and for how to activate the DDL licence.

7.1. Functions Overview

| Function | Short Description | Page |
|--|---|------|
| BOOL PI_DDL (int <i>ID</i> , int <i>iDdlTableId</i> , int <i>iOffsetOfFirstPointInDdlTable</i> , int <i>iNumberOfValues</i> , double* <i>pdValueArray</i>) | Transfer DDL data to a DDL data table on controller | 107 |
| BOOL PI_DPO (int <i>ID</i> , const char* <i>szAxes</i>) | Recalculate internal DDL processing parameters | 108 |
| BOOL PI_DTC (int <i>ID</i> , const int <i>piDdlTableIdsArray</i> , int <i>iArraySize</i>) | Clear given DDL table | 108 |
| BOOL PI_IsGeneratorRunning (int <i>ID</i> , const int* <i>piWaveGeneratorIdsArray</i> , BOOL* <i>pbValueArray</i> , int <i>iArraySize</i>) | Check if wave generators are running | 108 |
| BOOL PI_qDDL (int <i>ID</i> , const int* <i>piDdlTableIdsArrays</i> , int <i>iNumberOfDdlTables</i> , int <i>iOffset</i> , int <i>nrValues</i> , double** <i>pdValArray</i> , char* <i>szGcsArrayHeader</i> , int <i>iGcsArrayHeaderMaxSize</i>) | Read DDL tables asynchronously | 108 |
| BOOL PI_qDDL_SYNC (int <i>ID</i> , int <i>iDdlTableId</i> , int <i>iOffsetOfFirstPointInDdlTable</i> , int <i>iNumberOfValues</i> , double* <i>pdValueArray</i>) | Read DDL tables synchronously | 109 |
| BOOL PI_qDTL (int <i>ID</i> , const int* <i>piDdlTableIdsArray</i> , int* <i>piValueArray</i> , int <i>iArraySize</i>) | Get DDL table length | 109 |
| BOOL PI_qGWD (int <i>ID</i> , const int* <i>piWaveTableIdsArray</i> , int <i>iNumberOfWaveTables</i> , int <i>iOffset</i> , int <i>nrValues</i> , double** <i>pdValArray</i> , char* <i>szGcsArrayHeader</i> , int <i>iGcsArrayHeaderMaxSize</i>) | Read wave tables asynchronously | 110 |
| BOOL PI_qGWD_SYNC (int <i>ID</i> , int <i>iWaveTableId</i> , int <i>iOffsetOfFirstPointInWaveTable</i> , int <i>iNumberOfValues</i> , double* <i>pdValueArray</i>) | Read wave tables | 110 |
| BOOL PI_qTLT (int <i>ID</i> , int* <i>iNumberOfLinearizationTables</i>) | Get the number of DDL data tables | 110 |
| BOOL PI_qTWG (int <i>ID</i> , int* <i>iNumberOfWaveGenerators</i>) | Get the number of wave generators | 111 |
| BOOL PI_qTWS (int <i>ID</i> , const int* <i>piTriggerChannelIdsArrays</i> , int <i>iNumberOfTriggerChannels</i> , int <i>iOffset</i> , int <i>nrValues</i> , double** <i>pdValArray</i> , char* <i>szGcsArrayHeader</i> , int <i>iGcsArrayHeaderMaxSize</i>) | Read trigger line settings | 111 |
| BOOL PI_qWAV (int <i>ID</i> , const int* <i>piWaveTableIdsArray</i> , const int* <i>piParameterIdsArray</i> , double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Get the parameters for a defined waveform | 111 |
| BOOL PI_qWGC (int <i>ID</i> , const int* <i>piWaveGeneratorIdsArray</i> , int* <i>piValueArray</i> , int <i>iArraySize</i>) | Get the number of wave generator output cycles | 112 |
| BOOL PI_qWGO (int <i>ID</i> , const int <i>piWaveGeneratorIdsArray</i> , int* <i>piValueArray</i> , int <i>iArraySize</i>) | Get the start/stop mode of the given wave generator | 112 |
| BOOL PI_qWMS (int <i>ID</i> , const int* <i>piWaveTableIdsArray</i> , int* <i>iMaximumWaveSizeArray</i> , int <i>iArraySize</i>) | Gets the maximum size of the wave storage | 112 |
| BOOL PI_qWOS (int <i>ID</i> , const int* <i>piWaveGeneratorIdsArray</i> , double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Reads the current value of the offset which is added to the wave generator output | 112 |
| BOOL PI_qWSL (int <i>ID</i> , const int* <i>piWaveGeneratorIdsArray</i> , int* <i>piWaveTableIdsArray</i> , int <i>iArraySize</i>) | Get current setting of wave table selection | 113 |
| BOOL PI_qWTR (int <i>ID</i> , const int* <i>piWaveGeneratorIdsArray</i> , long* <i>piTableRateArray</i> , long* <i>piInterpolationTypeArray</i> , int <i>iArraySize</i>) | Gets the current wave generator table rate | 113 |
| BOOL PI_TWC (int <i>ID</i>) | Clears all output trigger settings for the wave generators | 113 |

| Function | Short Description | Page |
|---|--|------|
| BOOL PI_TWS (int <i>ID</i> , const int* <i>piTriggerChannelIdsArray</i> , const int* <i>piPointNumberArray</i> , const int* <i>piSwitchArray</i> , int <i>iArraySize</i>) | Sets trigger line actions to waveform points for the given trigger output line | 113 |
| BOOL PI_WAV_SINP (int <i>ID</i> , int <i>iWaveTableId</i> , int <i>iOffsetOfFirstPointInWaveTable</i> , int <i>iNumberOfPoints</i> , int <i>iAppendWave</i> , int <i>iCenterPointOfWave</i> , double <i>dAmplitudeOfWave</i> , double <i>dOffsetOfWave</i> , int <i>iSegmentLength</i>) | Define sine curve for given wave table | 114 |
| BOOL PI_WAV_LIN (int <i>ID</i> , int <i>iWaveTableId</i> , int <i>iOffsetOfFirstPointInWaveTable</i> , int <i>iNumberOfWavePoints</i> , int <i>iAppendWave</i> , int <i>iNumberOfSpeedUpDownPointsOfWave</i> , double <i>dAmplitudeOfWave</i> , double <i>dOffsetOfWave</i> , int <i>iSegmentLength</i>) | Define a single scan line curve for given wave table | 115 |
| BOOL PI_WAV_PNT (int <i>ID</i> , int <i>iWaveTableId</i> , int <i>iOffsetOfFirstPointInWaveTable</i> , int <i>iNumberOfWavePoints</i> , int <i>iAppendWave</i> , const double* <i>pdWavePoints</i>) | Create a user-defined curve for given wave table | 116 |
| BOOL PI_WAV_RAMP (int <i>ID</i> , int <i>iWaveTableId</i> , int <i>iOffsetOfFirstPointInWaveTable</i> , int <i>iNumberOfWavePoints</i> , int <i>iAppendWave</i> , int <i>iCenterPointOfWave</i> , int <i>iNumberOfSpeedUpDownPointsOfWave</i> , double <i>dAmplitudeOfWave</i> , double <i>dOffsetOfWave</i> , int <i>iSegmentLength</i>) | Define a ramp curve for given wave table | 116 |
| BOOL PI_WCL (int <i>ID</i> , int <i>iWaveTableIdsArray</i> , int <i>iArraySize</i>) | Clears the content of the given wave table | 117 |
| BOOL PI_WGC (int <i>ID</i> , const int* <i>piWaveGeneratorIdsArray</i> , const int* <i>piNumberOfCyclesArray</i> , int <i>iArraySize</i>) | Set the number of cycles for the wave generator output | 118 |
| BOOL PI_WGO (int <i>ID</i> , const int* <i>piWaveGeneratorIdsArray</i> , const int* <i>piStartModArray</i> , int <i>iArraySize</i>) | Start and stop the specified wave generator with the given mode | 118 |
| BOOL PI_WGR (int <i>ID</i>) | Restarts recording when the wave generator is running | 118 |
| BOOL PI_WOS (int <i>ID</i> , const int* <i>piWaveGeneratorIdsArray</i> , const double* <i>pdValueArray</i> , int <i>iArraySize</i>) | Sets an offset to the output of a wave generator | 118 |
| BOOL PI_WSL (int <i>ID</i> , const int* <i>piWaveGeneratorIdsArray</i> , const int* <i>piWaveTableIdsArray</i> , int <i>iArraySize</i>) | Wave table selection: connects a wave table to a wave generator | 119 |
| BOOL PI_WTR (int <i>ID</i> , const int* <i>piWaveGeneratorIdsArray</i> , const long* <i>piTableRateArray</i> , const long* <i>piInterpolationTypeArray</i> , int <i>iArraySize</i>) | Set wave generator table rate and interpolation type | 119 |

7.2. Function Documentation

BOOL PI_DDL (int *ID*, int *iDdlTableId*, int *iOffsetOfFirstPointInDdlTable*, int *iNumberOfValues*, double* *pdValueArray*)

Corresponding command: DDL

Transfer dynamic digital linearization feature data to a DDL data table on the controller.

Arguments:

ID ID of controller

iDdlTableId number of the DDL data table to use.

iOffsetOfFirstPointInDdlTable index of first value to be transferred, (the first value in the DDL table has index 1)

iNumberOfValues number of values to be transferred

pdValueArray Array with the values for the DDL table (can have been filled with PI_qDDL()).

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_DPO (int *ID*, const char* *szAxes*)

Corresponding command: DPO

Dynamic Digital Linearization (DDL) Parameter Optimization. Recalculates the internal DDL processing parameters (Time Delay Max, ID 0x14000006, Time Delay Min, ID 0x14000007) for specified axis.

Arguments:

ID ID of controller

szAxes string with axes, if "" or **NULL** all axes are affected.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_DTC (int *ID*, const int *piDdlTableIdsArray*, int *iArraySize*)

Corresponding command: DTC

Dynamic Digital Linearization (DDL) table clear: clears the given DDL table.

PI_DTC() also stops a running DDL initialization process.

Arguments:

ID ID of controller

piDdlTableIdsArray array with the IDs of the data tables which are to be cleared.

iArraySize the size of the array *piDdlTableIdsArray*.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_IsGeneratorRunning (int *ID*, const int* *piWaveGeneratorIdsArray*, **BOOL*** *pbValueArray*, int *iArraySize*)

Corresponding command: #9 (ASCII 9)

Check if wave generators are running. If **TRUE** for a wave generator, the corresponding element of the array will be set to **TRUE**, otherwise to **FALSE**. If no wave generators were specified, only one boolean value is set and it is placed in *pbValueArray[0]*: It is **TRUE** if at least one wave generator is **TRUE**, **FALSE** otherwise.

Arguments:

ID ID of controller

piWaveGeneratorIdsArray string with wave generator IDs, if "" or **NULL** all wave generators are queried and a global result placed in *pbValueArray[0]*

pbValueArray array to receive status of the wave generators, **TRUE** for wave generator in progress, **FALSE** otherwise

iArraySize the size of the array *pbValueArray* and *piWaveGeneratorIdsArray* (if not **NULL**)

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qDDL (int *ID*, const int* *piDdlTableIdsArrays*, int *iNumberOfDdlTables*, int *iOffset*, int *nrValues*, double** *pdValArray*, char* *szGcsArrayHeader*, int *iGcsArrayHeaderMaxSize*)

Corresponding command: DDL?

Read DDL tables. This function reads the data asynchronously, it will return as soon as the data header has been read and start a background process which reads in the data itself. See PI_GetAsyncBuffer() and PI_GetAsyncBufferIndex(). Detailed information about the data read in can be found in the header sent by the controller. See the GCS Array manual for details.

Only tables with the same length can be read at the same time. Because DDL tables do not have a common length, use PI_qDDL to read the table length before reading the table data.

Arguments:

ID ID of controller
piDdlTableIdsArray IDs of DDL tables
iNumberOfDdlTables number of DDL tables to read
iOffset index of first value to be read (starts with index 1)
nrValues number of values to read
pdValarray pointer to internal array to store the data; data from all tables read will be placed in the same array with the values interspersed; the DLL will allocate enough memory to store all data, call **PI_GetAsyncBufferIndex()** to find out how many data points have already been transferred
szGcsArrayHeader buffer to store the GCS array header
iGcsArrayHeaderMaxSize size of the buffer to store the GCS array header, must be given to prevent buffer overflow

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qDDL_SYNC (int *ID*, int *iDdlTableId*, int *iOffsetOfFirstPointInDdlTable*, int *iNumberOfValues*, double* *pdValueArray*)

Corresponding command: DDL?

Get the dynamic digital linearization feature data from a DDL data table from the controller. For large *N* values, communication timeout must be set long enough, otherwise a communication error may occur.

Arguments:

ID ID of controller
iDdlTableId ID of the DDL data table.
iOffsetOfFirstPointInDdlTable index in the DDL table of first value to be read, the first value in the DDL table has index 1
iNumberOfValues number of values to be read
pdValueArray Array to receive the values. Caller is responsible for providing enough space for *iNumberOfValues* doubles

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qDTL (int *ID*, const int* *piDdlTableIdsArray*, int* *piValueArray*, int *iArraySize*)

Corresponding command: DTL?

Get Dynamic Digital Linearizations Table Length.
The table length should be read before reading the table data by PI_qDDL.

Arguments:

ID ID of controller
piDdlTableIdsArray array of the DDL table IDs
piValueArray array to receive the DDL table size
iArraySize the size of the arrays *piDdlTableIdsArray* and *piValueArray*

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

```

BOOL PI_qGWD (int ID, const int* piWaveTableIdsArrays, int
iNumberOfWaveTables, int iOffset, int nrValues, double** pdValArray, char*
szGcsArrayHeader, int iGcsArrayHeaderMaxSize)

```

Corresponding command: GWD?

Read wave tables. This function reads the data asynchronously, it will return as soon as the data header has been read and start a background process which reads in the data itself. See `PI_GetAsyncBuffer()` and `PI_GetAsyncBufferIndex()`. Detailed information about the data read in can be found in the header sent by the controller. See the GCS Array manual for details.

Depending on the waveform definition with `PI_WAV()`, the wave tables may have different lengths. But due to the definition of the GCS array as the response format, it is not possible to read tables with different lengths at the same time. You can ask with `PI_qWAV()` for the current length of the wave tables.

Arguments:

ID ID of controller

piWaveTableIdsArray IDs of wave tables

iNumberOfWaveTables number of wave tables to read

iOffset index of first value to be read (starts with index 1)

nrValues number of values to read

pdValarray pointer to internal array to store the data; data from all tables read will be placed in the same array with the values interspersed; the DLL will allocate enough memory to store all data, call `PI_GetAsyncBufferIndex()` to find out how many data points have already been transferred

szGcsArrayHeader buffer to store the GCS array header

iGcsArrayHeaderMaxSize size of the buffer to store the GCS array header, must be given to prevent buffer overflow

Returns:

TRUE if successful, **FALSE** otherwise

```

BOOL PI_qGWD_SYNC (int ID, int iWaveTableId, int
iOffsetOfFirstPointInWaveTable, int iNumberOfValues, double* pdValueArray )

```

Corresponding command: GWD?

Read the waveform associated with *iWaveTableId*.

Arguments:

ID ID of controller

iWaveTableId identifier for wave table

iOffsetOfFirstPointInWaveTable index of first point to be read, starts with index 1

iNumberOfPoints number of points to read

pdValuesArray array to receive the waveform. (Caller must provide enough space to store *iNumberOfPoints* double values!)

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

```

BOOL PI_qTLT (int ID, int* iNumberOfLinearizationTables )

```

Corresponding command: TLT?

Get the number of DDL data tables.

Arguments:

ID ID of controller

piNumberOfLinearizationTables pointer to receive the number of DDL data tables.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qTWG (int *ID*, int* *iNumberOfWaveGenerators*)

Corresponding command: TWG?

Get the number of wave generators.

Arguments:

ID ID of controller

iNumberOfWaveGenerators pointer to store the number of wave generators.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qTWS (int *ID*, const int* *piTriggerChannelIdsArrays*, int *iNumberOfTriggerChannels*, int *iOffset*, int *nrValues*, double** *pdValArray*, char* *szGcsArrayHeader*, int *iGcsArrayHeaderMaxSize*)

Corresponding command: TWS?

Reading of the trigger line settings made with PI_TWS for the waveform points. This function reads the data asynchronously, it will return as soon as the data header has been read and start a background process which reads in the data itself. See PI_GetAsyncBuffer() and PI_GetAsyncBufferIndex(). Detailed information about the data read in can be found in the header sent by the controller. See the GCS Array manual for details.

To query the waveform shape, use PI_qGWD.

Arguments:

ID ID of controller

piTriggerChannelIdsArray IDs of the trigger line (digital output line)

iNumberOfTriggerChannels number of trigger lines to read

iOffset index of first value to be read (starts with index 1)

nrValues number of values to read

pdValarray pointer to internal array to store the data; data from all trigger lines read will be placed in the same array with the values interspersed; the DLL will allocate enough memory to store all data, call PI_GetAsyncBufferIndex() to find out how many data points have already been transferred

szGcsArrayHeader buffer to store the GCS array header

iGcsArrayHeaderMaxSize size of the buffer to store the GCS array header, must be given to prevent buffer overflow

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qWAV (int *ID*, const int* *piWaveTableIdsArray*, const int* *piParameterIdsArray*, double* *pdValueArray*, int *iArraySize*)

Corresponding command: WAV?

Get the parameters for a defined waveform. For each desired parameter you must specify a wave table in *piWaveTableIdsArray* and a parameter ID in the corresponding element of *piParameterIdsArray*. The following parameter ID is valid:

1: Number of waveform points for currently defined wave.

Arguments:

ID ID of controller

piWaveTableIdsArray array with wave table IDs for which the parameter(s) should be read

piParameterIdsArray array with IDs of requested parameters

pdValueArray array to be filled with the values for the parameters

iArraySize the size of the arrays *piWaveTablesArray*, *piParameterIdsArray*, and *pdValueArray*

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qWGC (int *ID*, const int* *piWaveGeneratorIdsArray*, int* *piValueArray*, int *iArraySize*)

Corresponding command: WGC?

Get the number of wave generator output cycles set by PI_WGC (p. 118).

Arguments:

ID ID of controller

piWaveGeneratorIdsArray array with wave generators, if **NULL** all wave generators are queried

piValueArray array with number of cycles for each wave generator in *piWaveGeneratorIdsArray*

iArraySize the size of the arrays *piWaveGeneratorIdsArray* (if not **NULL**) and *piValueArray*.

Returns:

TRUE if no error, **FALSE** otherwise

BOOL PI_qWGO (int *ID*, const int *piWaveGeneratorIdsArray*, int* *piValueArray*, int *iArraySize*)

Corresponding command: WGO?

Get the start/stop mode of the given wave generator.

Note: Ask with PI_IsGeneratorRunning() for the current activation state of the wave generator. The reply shows if a wave generator is running or not, but does not contain any information about the wave generator start mode (e.g. with trigger output). With PI_qWGO you can ask for the last commanded wave generator start option (set by PI_WGO (p. 118)).

Arguments:

ID ID of controller

piWaveGeneratorIdsArray array with wave generators for which the start mode values will be read out, if **NULL** all wave generators are queried

piValueArray array with modes for each wave generator in *piWaveGeneratorIdsArray*

iArraySize the size of the arrays *piWaveGeneratorIdsArray* (if not **NULL**) and *piValueArray*.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qWMS (int *ID*, const int* *piWaveTableIdsArray*, int* *iMaximumWaveSizeArray*, int *iArraySize*)

Corresponding command: WMS?

Gets the maximum size of the wave storage for *piWaveTableIdsArray*

Arguments:

ID ID of controller

piWaveTableIdsArray array with wave tables, if **NULL** all wave tables are queried.

piMaximumWaveSizeArray array to be filled with the maximum size of the wave storage for the corresponding wave table (number of points).

iArraySize the size of the arrays *piWaveTableIdsArray* (if not **NULL**) and *piMaximumWaveSizeArray*.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qWOS (int *ID*, const int* *piWaveGeneratorIdsArray*, double* *pdValueArray*, int *iArraySize*)

Corresponding command: WOS?

Reads the current value of the offset which is added to the wave generator output.

Arguments:

ID ID of controller

piWaveGeneratorIdsArray array with wave generators, if **NULL** all wave generators are queried.

pdValueArray array to receive the offsets of the wave generators.

iArraySize the size of the arrays *piWaveGeneratorIdsArray* (if not **NULL**) and *pdValueArray*.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qWSL (int *ID*, const int* *piWaveGeneratorIdsArray*, int* *piWaveTableIdsArray*, int *iArraySize*)

Corresponding command: WSL?

Get current setting of wave table selection

Arguments:

ID ID of controller

piWaveGeneratorIdsArray array with wave generators, if **NULL** all wave generators are queried.

piWaveTableIdsArray array to receive the wave table IDs

iArraySize the size of the arrays *piWaveGeneratorIdsArray* (if not **NULL**) and *piWaveTableIdsArray*.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_qWTR (int *ID*, const int* *piWaveGeneratorIdsArray*, long* *piTableRateArray*, long* *piInterpolationTypeArray*, int *iArraySize*)

Corresponding command: WTR?

Gets the current wave generator table rate, i.e. the number of servo-loop cycles used for wave generator output. Gets also the interpolation type used with table rate values > 1.

Arguments:

ID ID of controller

piWaveGeneratorIdsArray array with wave generators, if **NULL** all wave generators are queried.

piTableRateArray array to receive the wave table rate.

piInterpolationTypeArray array to receive the interpolation type.

iArraySize the size of the arrays *piWaveGeneratorIdsArray* (if not **NULL**) and *piTableRateArray* and *piInterpolationTypeArray*.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_TWC (int *ID*)

Corresponding command: TWC

Clears all output trigger settings for the wave generators (the settings made with PI_TWS).

Arguments:

ID ID of controller

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_TWS (int *ID*, const int* *piTriggerChannelIdsArray*, const int* *piPointNumberArray*, const int* *piSwitchArray*, int *iArraySize*)

Corresponding command: TWS

Sets trigger line actions to waveform points for the given trigger output line.

The power-on default state of all points is low. Afterwards, the signal state of the trigger output line can be switched to "low" for all points using PI_TWC(). It is recommended to use PI_TWC() before trigger actions are set with PI_TWS().

For the selected trigger output line the generator trigger mode must be activated by PI_CTO().

Arguments:

ID ID of controller

piTriggerChannelIdsArray array with the trigger output lines.

piPointNumberArray array with the wave points.

piSwitchArray with the signal states of the trigger output lines at the wave points, if zero the trigger is set low, otherwise the trigger is set high.

iArraySize the number of points in the arrays *piTriggerChannelIdsArray*, *piPointNumberArray* and *piSwitchArray*.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

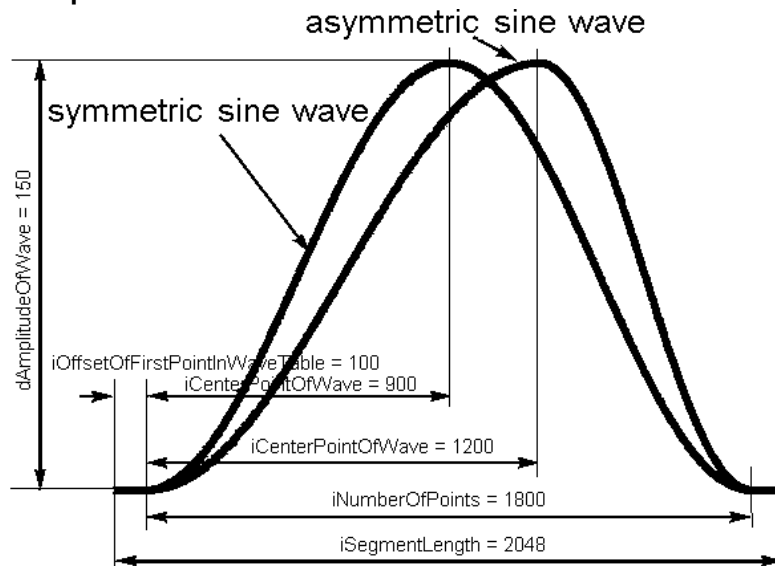
BOOL PI_WAV_SIN_P (int *ID*, int *iWaveTableId*, int *iOffsetOfFirstPointInWaveTable*, int *iNumberOfPoints*, int *iAppendWave*, int *iCenterPointOfWave*, double *dAmplitudeOfWave*, double *dOffsetOfWave*, int *iSegmentLength*)

Corresponding command: WAV

Define sine curve for given wave table.

To allow for flexible waveform shapes, a wave table can be divided into "segments". Each segment must be defined with a separate call of PI_WAV_SIN_P() or one of the other PI_WAV functions. In doing so, the *iAppendWave* argument (see below) is used to concatenate the segments so that they will form the final waveform. To change individual segments or to modify their order, the complete waveform must be recreated segment-by-segment. See the user manual of the controller for more information and for more examples.

Example:



Arguments:

ID ID of controller

iWaveTableId The wave table ID

iOffsetOfFirstPointInWaveTable The index of the starting point of the curve in the segment. Gives the phase shift. Lowest possible value is 0.

iNumberOfPoints The length of the curve as number of points.

iAppendWave Possible values (supported values depend on controller):

0 = clears the wave table and starts writing with the first point in the table

1 = adds the content of the defined segment to the already existing wave table contents (i.e. the values of the defined points are added to the existing values of that points)

2 = appends the defined segment to the already existing wave table content (i.e. concatenates segments to form one final waveform)

iCenterPointOfWave The index of the center point of the sine curve. Determines if the curve is symmetrically or not. Lowest possible value is 0.

dAmplitudeOfWave The amplitude of the sine curve.

dOffsetOfWave The offset of the sine curve.

iSegmentLength The length of the wave table segment as number of points. Only the number of points given by *iSegmentLength* will be written to the wave table. If the *iSegmentLength* value is

larger than the *iNumberOfPoints* value, the missing points in the segment are filled up with the endpoint value of the curve.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

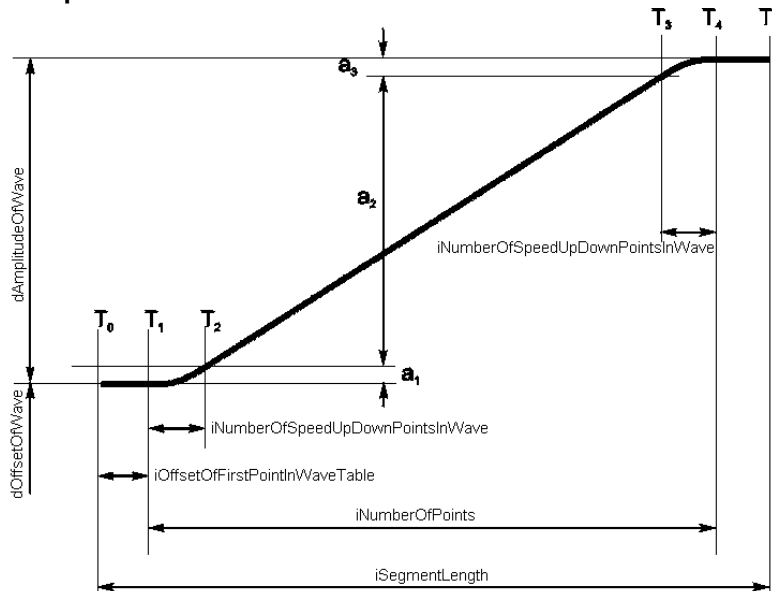
BOOL PI_WAV_LIN (int *ID*, int *iWaveTableId*, int *iOffsetOfFirstPointInWaveTable*, int *iNumberOfWavePoints*, int *iAppendWave*, int *iNumberOfSpeedUpDownPointsOfWave*, double *dAmplitudeOfWave*, double *dOffsetOfWave*, int *iSegmentLength*)

Corresponding command: WAV

Define a single scan line curve for given wave table.

To allow for flexible waveform shapes, a wave table can be divided into "segments". Each segment must be defined with a separate call of PI_WAV_LIN() or one of the other PI_WAV functions. In doing so, the *iAppendWave* argument (see below) is used to concatenate the segments so that they will form the final waveform. To change individual segments or to modify their order, the complete waveform must be recreated segment-by-segment. See the user manual of the controller for more information and for more examples.

Example:



Arguments:

ID ID of controller

iWaveTableId The ID of the wave table

iOffsetOfFirstPointInWaveTable The index of the starting point of the scan line in the segment.
Lowest possible value is 0.

iNumberOfWavePoints The length of the single scan line curve as number of points.

iAppendWave Possible values (supported values depend on controller):

0 = clears the wave table and starts writing with the first point in the table

1 = adds the content of the defined segment to the already existing wave table contents (i.e. the values of the defined points are added to the existing values of that points)

2 = appends the defined segment to the already existing wave table content (i.e. concatenates segments to form one final waveform)

dAmplitudeOfWave The amplitude of the scan line.

iNumberOfSpeedUpDownPointsOfWave The number of points for speed up and down.

dOffsetOfWave The offset of the scan line

iSegmentLength The length of the wave table segment as number of points. Only the number of points given by *iSegmentLength* will be written to the wave table. If the *iSegmentLength* value is larger than the *iNumberOfPoints* value, the missing points in the segment are filled up with the endpoint value of the curve.

Returns:

TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_WAV_PNT (int *ID*, int *iWaveTableId*, int *iOffsetOfFirstPointInWaveTable*, int *iNumberOfWavePoints*, int *iAppendWave*, const double* *pdWavePoints*)

Corresponding command: WAV

Create a user-defined curve for given wave table.

To allow for flexible waveform shapes, a wave table can be divided into "segments". Each segment must be defined with a separate call of PI_WAV_PNT() or one of the other PI_WAV functions. In doing so, the *iAppendWave* argument (see below) is used to concatenate the segments so that they will form the final waveform. To change individual segments or to modify their order, the complete waveform must be recreated segment-by-segment. See the user manual of the controller for more information and for more examples.

Arguments:

ID ID of controller

iWaveTableId The ID of the wave table

iOffsetOfFirstPointInWaveTable The index of the starting point. Must be 1.

iNumberOfWavePoints The length of the user-defined curve as number of points.

iAppendWave Possible values (supported values depend on controller):

0 = clears the wave table and starts writing with the first point in the table

1 = adds the content of the defined segment to the already existing wave table contents (i.e. the values of the defined points are added to the existing values of that points)

2 = appends the defined segment to the already existing wave table content (i.e. concatenates segments to form one final waveform)

iSegmentLength The segment length, i.e. the number of points written to the wave table. Is identical to the *iNumberOfWavePoints* value.

pdWavePoints array with the wave points.

Returns:

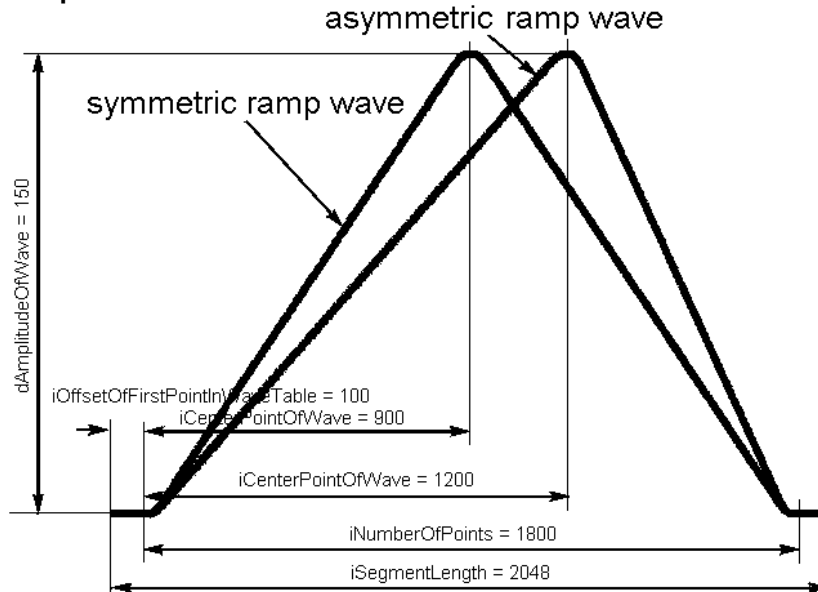
TRUE if no error, **FALSE** otherwise (see p. 7)

BOOL PI_WAV_RAMP (int *ID*, int *iWaveTableId*, int *iOffsetOfFirstPointInWaveTable*, int *iNumberOfWavePoints*, int *iAppendWave*, int *iCenterPointOfWave*, int *iNumberOfSpeedUpDownPointsOfWave*, double *dAmplitudeOfWave*, double *dOffsetOfWave*, int *iSegmentLength*)

Corresponding command: WAV

Define a ramp curve for given wave table.

To allow for flexible waveform shapes, a wave table can be divided into "segments". Each segment must be defined with a separate call of PI_WAV_RAMP() or one of the other PI_WAV functions. In doing so, the *iAppendWave* argument (see below) is used to concatenate the segments so that they will form the final waveform. To change individual segments or to modify their order, the complete waveform must be recreated segment-by-segment. See the user manual of the controller for more information and for more examples.

Example:**Arguments:****ID** ID of controller**iWaveTableId** The ID of the wave table**iOffsetOfFirstPointInWaveTable** The index of the starting point of the ramp curve in the segment.
Gives the phase shift. Lowest possible value is 0.**iNumberOfWavePoints** The length of the ramp curve as number of points.**iAppendWave** Possible values (supported values depend on controller):

0 = clears the wave table and starts writing with the first point in the table

1 = adds the content of the defined segment to the already existing wave table contents (i.e. the values of the defined points are added to the existing values of that points)

2 = appends the defined segment to the already existing wave table content (i.e. concatenates segments to form one final waveform)

iCenterPointOfWave The index of the center point of the ramp curve. Determines if the curve is symmetrically or not. Lowest possible value is 0.**dAmplitudeOfWave** The amplitude of the ramp curve.**iNumberOfSpeedUpDownPointsOfWave** The number of points for speed up and down.**dOffsetOfWave** The offset of the ramp curve.**iSegmentLength** The length of the wave table segment as number of points. Only the number of points given by *iSegmentLength* will be written to the wave table. If the *iSegmentLength* value is larger than the *iNumberOfPoints* value, the missing points in the segment are filled up with the endpoint value of the curve.**Returns:****TRUE** if no error, **FALSE** otherwise (see p. 7)**BOOL PI_WCL** (int *ID*, int *iWaveTableId*Array, int *iArraySize*)**Corresponding command: WCL**

Clears the content of the given wave table.

As long as a wave generator is running, it is not possible to delete the connected wave table.

Arguments:**ID** ID of controller**iWaveTableId**Array array with the IDs of the wave tables to be cleared.**iArraySize** the size of the array *iWaveTableId*Array.**Returns:****TRUE** if no error, **FALSE** otherwise (see p. 7)

BOOL PI_WGC (int *ID*, const int* *piWaveGeneratorIdsArray*, const int* *piNumberOfCyclesArray*, int *iArraySize*)

Corresponding command: WGC

Set the number of cycles for the wave generator output (which is started with PI_WGO()).

Arguments:

ID ID of controller

piWaveGeneratorIdsArray array with wave generators

piNumberOfCyclesArray array with number of cycles for each wave generator in *piWaveGeneratorIdsArray*

iArraySize the size of the arrays *piWaveGeneratorIdsArray* and *piNumberOfCyclesArray*

Returns:

TRUE if successful, FALSE otherwise

BOOL PI_WGO (int *ID*, const int* *piWaveGeneratorIdsArray*, const int* *piStartModArray*, int *iArraySize*)

Corresponding command: WGO

Start and stop the specified wave generator with the given mode. Depending on the controller, starts also data recording.

Arguments:

ID ID of controller

piWaveGeneratorIdsArray array with wave generators.

piStartModArray array with start modes for each wave generator in *piWaveGeneratorIdsArray* (hex format, optional decimal format)

iArraySize the size of the arrays *piWaveGeneratorIdsArray* and *piStartModArray*

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_WGR (int *ID*)

Corresponding command: WGR

Restarts recording when the wave generator is running (a first data recording cycle is started with PI_WGO() which starts the wave generator output).

The data recorder configuration can be made with PI_DRC() and PI_DRT. Data can be read with PI_qDRR() (p. 68).

Arguments:

ID ID of controller

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_WOS (int *ID*, const int* *piWaveGeneratorIdsArray*, const double* *pdValueArray*, int *iArraySize*)

Corresponding command: WOS

Sets an offset to the output of a wave generator. The current wave generator output is then created by adding the offset value to the current wave value:

Generator Output = Offset + Current Wave Value

Do not confuse the output-offset value set with PI_WOS() with the offset settings done during the waveform creation with the PI_WAV() functions. While the PI_WAV() offset belongs to only one waveform, the PI_WOS() offset is added to all waveforms which are output by the given wave generator.

Deleting wave table content with PI_WCL() has no effect on the offset settings for the wave generator output.

Arguments:

ID ID of controller

piWaveGeneratorIdsArray array with wave generators.

pdValueArray array with the offsets of the wave generators.

iArraySize the size of the arrays *piWaveGeneratorIdsArray* and *pdValueArray*.

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_WSL (int *ID*, const int* *piWaveGeneratorIdsArray*, const int* *piWaveTableIdsArray*, int *iArraySize*)

Corresponding command: WSL

Wave table selection: connects a wave table to a wave generator or disconnects the selected generator from any wave table.

Two or more generators can be connected to the same wave table, but a generator cannot be connected to more than one wave table.

Deleting wave table content with PI_WCL has no effect on the PI_WSL settings.

As long as a wave generator is running, it is not possible to change the connected wave table.

Arguments:

ID ID of controller

piWaveGeneratorIdsArray array with wave generators.

piWaveTableIdsArray array with the wave table ID. "0" disconnects the selected generator from any wave table.

iArraySize the size of the arrays *piWaveGeneratorIdsArray* and *piWaveTableIdsArray*.

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

BOOL PI_WTR (int *ID*, const int* *piWaveGeneratorIdsArray*, const long* *piTableRateArray*, const long* *piInterpolationTypeArray*, int *iArraySize*)

Corresponding command: WTR

Set wave generator table rate and interpolation type:

Using PI_WTR(), you can "extend" the individual output cycles of the waveform. Depending on the controller, PI_WTR() furthermore determines the type of interpolation to use for the wave generator output. If the Wave Generator Table Rate is larger than 1, an interpolation helps to avoid sudden position jumps of the axis controlled by the wave generator.

Arguments:

ID ID of controller

piWaveGeneratorIdsArray array with wave generator IDs.

piTableRateArray array with the wave table rates.

piInterpolationTypeArray array with the interpolation types.

iArraySize the size of the arrays *piWaveGeneratorIdsArray* and *piTableRateArray* and *piInterpolationTypeArray*.

Returns:

TRUE if no error, FALSE otherwise (see p. 7)

8. Functions for User-Defined Stages

8.1. Overview

The information in this chapter is relevant for controllers which support the use of stage databases that are located on the host PC. See the user manual of your controller to find out if the use of such stage databases is supported.

The PI GCS 2 DLL has functions allowing you to both define and save new stages (parameter sets) to the PI_UserStages2.dat stage database on the host PC (see “Stage Databases” below for more information). Being able to specify the parameters of a stage and then save those parameters as a set under the stage name makes it easier to connect to previously defined stages.

You can create a new stage parameter set by changing the stage parameters with PI_SPA(). It is important to set the stage parameters correctly. Note that the parameter which determines whether a stage is “new” or not is the Stage Name parameter (ID 0x3C). If it is not specified, the PI_AddStage command will fail. See the user manual of the controller for a complete parameter list and parameter handling details.

You can ease the creation by loading an existing parameter set with PI_CST() and afterwards change the name and any other parameters, which differ, with PI_SPA(). PI_CST() “connects” a valid stage, i.e. makes its parameter set active. It uses the corresponding parameters in the DAT files, so that you do not have to set them all by yourself.

To save the new stage and thus make it available for a future connection with PI_CST(), use PI_AddStage() to add its parameter set to PI_UserStages2.dat. After adding it to PI_UserStages2.dat, the stage will also appear in the list returned by PI_qVST().

If you want to remove a stage from PI_UserStages2.dat call PI_RemoveStage().

It may be more comfortable to set the stage parameters using the PIStageEditor (a GUI dialog). See the separate PI Stage Editor manual (SM144E) for a description of how to operate that graphic interface.

The PIStageEditor can also be started from PIMikroMove. This program provides several functions which ease creating and editing stage parameter sets. For further information, refer to “Stage Editor” and “Tutorials - Frequently Asked Questions” in the PIMikroMove manual.

NOTES

The PI_OpenUserStagesEditDialog() or PI_OpenPiStagesEditDialog() functions are provided for compatibility reasons only and should not be used to open the PIStageEditor. Since the PIStageEditor is not modal, problems can occur when the calling application exits before the PIStageEditor window is closed. Please start the PIStageEditor either from PIMikroMove or via its executable.

```

BOOL PI_FUNC_DECL PI_AddStage (const int ID, char *const szAxes)
BOOL PI_FUNC_DECL PI_RemoveStage (const int ID, char *szStageName)

```


8.2. Function Description

BOOL PI_AddStage (const int *ID*, const char* *szAxes*)

Adds the stage specified for *szAxes* to the *PI_UserStages2.dat* file which contains user-defined stages.

Arguments:

ID ID of controller

szAxes string with axis identifier.

Returns:

TRUE if successful, **FALSE**, if the buffer was too small to store the message

BOOL PI_RemoveStage (const int *ID*, char * *szStageName*)

Removes the stage with the given name from the *PI_UserStages2.dat* file which contains user-defined stages.

Arguments:

ID ID of controller

szStageName the stage name as string.

Returns:

TRUE if successful, **FALSE**, if the buffer was too small to store the message

8.3. Stage Databases

The PI GCS 2 DLL and the GCS-based host software from PI use multiple databases for stage parameters:

- **PIStages2.dat** and **PIMicosStages2.dat** contain parameter sets for all standard stages from PI and PI miCos and are automatically installed on the host PC with the setup. They cannot be edited; should changes in a file become necessary, you must obtain a new version from PI or PI miCos and install it on your host PC.
- **PI_UserStages2.dat** allows you to create and save your own stages (see “Overview” on p. 120). This database is created on the host PC the first time you connect stages in the host software (i.e. the first time the *PI_qVST()* or *PI_CST()* functions of the PI GCS 2 library are used which is the case, for example, when *VST?* or *CST* are sent in *PITerminal* or the *Select connected stages* startup step is performed in *PIMikroMove*).
- **X-xxx.dat** files contain parameter sets for custom stages delivered by PI or PI miCos. Those files are provided with the stages and have to be copied to the host PC according to the accompanying instructions.
X-xxx.dat files cannot be edited; should changes become necessary, you must obtain a new version from PI or PI miCos.

The *PIStages2.dat*, *PIMicosStages2.dat*, *PI_UserStages2.dat* and *X-xxx.dat* databases are located in the ...*PI\GcsTranslator* directory on the host PC. The location of the *PI* directory is that specified upon installation, usually in *C:\Documents and Settings\All Users\Application Data* (Windows XP) or *C:\ProgramData* (Windows Vista, 7). If this directory does not exist, the program that needs the stage databases will look in its own directory. In *PIMikroMove*, you can use the *Version Info* item in the controller menu or the *Search for controller software* item in the *Connections* menu to identify the *GcsTranslator* path.

See the user manual of your controller for how to install or update stage databases on the host PC.

Notes for users which have already installed older versions of PI GCS 2 DLL, PIMikroMove and PIStageEditor:

- The format of the stage database files has changed (more parameters provided), realized by a file version change from 1 to 2. Note that stage database files with version 2 contain a "2" in their file name, e.g. PIStages2.dat (instead of PIStages.dat for version 1).
- Existing PI_Userstages DAT files of version 1 are automatically converted to version 2 files the first time you connect stages in the host software, i.e. the first time the PI_qVST() or PI_CST() functions of the PI GCS 2 library are used which is the case, for example, when VST? or CST are sent in PITerminal or the *Select connected stages* startup step is performed in PIMikroMove. Parameters which were not present in version 1 are set to default values during conversion.
- Version 4 and newer of the PIStageEditor supports stage database files of version 2 (in PIMikroMove, you can check the version of the PIStageEditor with *Help → Show version information...*).

8.4. Troubleshooting

Problem:

Stage database file cannot be opened, or stage selection in host software is not possible. Error message arises saying that the stage database does not have the correct revision.

Solution:

To support new hardware (controller or stages), it is necessary to release new revisions of the stage database files. Although PI aims for highest compatibility, the latest host software may not be able to work with older stage database files. You can check the revision of your stage database files using the *PIStageEditor* (see the PIStageEditor manual for details).

If your *PIStages2.dat* file does not have the correct revision, download the latest revision from www.pi.ws. For detailed update instructions see the user manual of the controller.

The *PI_UserStages2.dat* file is created the first time you connect stages in the host software (i.e. the first time the PI_qVST() or PI_CST() functions of the PI GCS 2 DLL are used). If you have already a *PI_UserStages2.dat* file for your controller but this file cannot be opened with the latest software, proceed as follows:

- 1 Rename the existing *PI_UserStages2.dat* file on your host PC.
- 2 Create a new *PI_UserStages2.dat*. This can be done by calling the PI_qVST() or PI_qCST() functions of the PI GCS 2 DLL.
- 3 Open the new *PI_UserStages2.dat* in the *PIStageEditor*.
- 4 Import the content of the old (renamed) *PI_UserStages2.dat* file to the new file. See the *PIStageEditor* manual for details. Note that during the import procedure, the imported stage parameter sets are converted to fit the new revision. Parameters which were not present in the old revision are set to default values which may need to be optimized.

9. Error Codes

The error codes listed here are those of the *PI General Command Set*. As such, some are not relevant to your controller and will simply never occur with the systems this manual describes.

Controller errors

| | | |
|----|--------------------------------------|---|
| 0 | PI_CNTR_NO_ERROR | No error |
| 1 | PI_CNTR_PARAM_SYNTAX | Parameter syntax error |
| 2 | PI_CNTR_UNKNOWN_COMMAND | Unknown command |
| 3 | PI_CNTR_COMMAND_TOO_LONG | Command length out of limits or command buffer overrun |
| 4 | PI_CNTR_SCAN_ERROR | Error while scanning |
| 5 | PI_CNTR_MOVE_WITHOUT_REF_OR_NO_SERVO | Unallowable move attempted on unreferenced axis, or move attempted with servo off |
| 6 | PI_CNTR_INVALID_SGA_PARAM | Parameter for SGA not valid |
| 7 | PI_CNTR_POS_OUT_OF_LIMITS | Position out of limits |
| 8 | PI_CNTR_VEL_OUT_OF_LIMITS | Velocity out of limits |
| 9 | PI_CNTR_SET_PIVOT_NOT_POSSIBLE | Attempt to set pivot point while U,V and W not all 0 |
| 10 | PI_CNTR_STOP | Controller was stopped by command |
| 11 | PI_CNTR_SST_OR_SCAN_RANGE | Parameter for SST or for one of the embedded scan algorithms out of range |
| 12 | PI_CNTR_INVALID_SCAN_AXES | Invalid axis combination for fast scan |
| 13 | PI_CNTR_INVALID_NAV_PARAM | Parameter for NAV out of range |
| 14 | PI_CNTR_INVALID_ANALOG_INPUT | Invalid analog channel |
| 15 | PI_CNTR_INVALID_AXIS_IDENTIFIER | Invalid axis identifier |
| 16 | PI_CNTR_INVALID_STAGE_NAME | Unknown stage name |
| 17 | PI_CNTR_PARAM_OUT_OF_RANGE | Parameter out of range |
| 18 | PI_CNTR_INVALID_MACRO_NAME | Invalid macro name |
| 19 | PI_CNTR_MACRO_RECORD | Error while recording macro |
| 20 | PI_CNTR_MACRO_NOT_FOUND | Macro not found |
| 21 | PI_CNTR_AXIS_HAS_NO_BRAKE | Axis has no brake |
| 22 | PI_CNTR_DOUBLE_AXIS | Axis identifier specified more than once |
| 23 | PI_CNTR_ILLEGAL_AXIS | Illegal axis |
| 24 | PI_CNTR_PARAM_NR | Incorrect number of parameters |
| 25 | PI_CNTR_INVALID_REAL_NR | Invalid floating point number |
| 26 | PI_CNTR_MISSING_PARAM | Parameter missing |
| 27 | PI_CNTR_SOFT_LIMIT_OUT_OF_RANGE | Soft limit out of range |
| 28 | PI_CNTR_NO_MANUAL_PAD | No manual pad found |
| 29 | PI_CNTR_NO_JUMP | No more step-response values |
| 30 | PI_CNTR_INVALID_JUMP | No step-response values recorded |
| 31 | PI_CNTR_AXIS_HAS_NO_REFERENCE | Axis has no reference sensor |
| 32 | PI_CNTR_STAGE_HAS_NO_LIM_SWITCH | Axis has no limit switch |
| 33 | PI_CNTR_NO_RELAY_CARD | No relay card installed |
| 34 | PI_CNTR_CMD_NOT_ALLOWED_FOR_STAGE | Command not allowed for selected stage(s) |
| 35 | PI_CNTR_NO_DIGITAL_INPUT | No digital input installed |

| | | |
|----|-------------------------------------|--|
| 36 | PI_CNTR_NO_DIGITAL_OUTPUT | No digital output configured |
| 37 | PI_CNTR_NO_MCM | No more MCM responses |
| 38 | PI_CNTR_INVALID_MCM | No MCM values recorded |
| 39 | PI_CNTR_INVALID_CNTR_NUMBER | Controller number invalid |
| 40 | PI_CNTR_NO_JOYSTICK_CONNECTED | No joystick configured |
| 41 | PI_CNTR_INVALID_EGE_AXIS | Invalid axis for electronic gearing, axis cannot be slave |
| 42 | PI_CNTR_SLAVE_POSITION_OUT_OF_RANGE | Position of slave axis is out of range |
| 43 | PI_CNTR_COMMAND_EGE_SLAVE | Slave axis cannot be commanded directly when electronic gearing is enabled |
| 44 | PI_CNTR_JOYSTICK_CALIBRATION_FAILED | Calibration of joystick failed |
| 45 | PI_CNTR_REFERENCING_FAILED | Referencing failed |
| 46 | PI_CNTR_OPM_MISSING | OPM (Optical Power Meter) missing |
| 47 | PI_CNTR_OPM_NOT_INITIALIZED | OPM (Optical Power Meter) not initialized or cannot be initialized |
| 48 | PI_CNTR_OPM_COM_ERROR | OPM (Optical Power Meter) communication error |
| 49 | PI_CNTR_MOVE_TO_LIMIT_SWITCH_FAILED | Move to limit switch failed |
| 50 | PI_CNTR_REF_WITH_REF_DISABLED | Attempt to reference axis with referencing disabled |
| 51 | PI_CNTR_AXIS_UNDER_JOYSTICK_CONTROL | Selected axis is controlled by joystick |
| 52 | PI_CNTR_COMMUNICATION_ERROR | Controller detected communication error |
| 53 | PI_CNTR_DYNAMIC_MOVE_IN_PROCESS | MOV! motion still in progress |
| 54 | PI_CNTR_UNKNOWN_PARAMETER | Unknown parameter |
| 55 | PI_CNTR_NO_REP_RECORDED | No commands were recorded with REP |
| 56 | PI_CNTR_INVALID_PASSWORD | Password invalid |
| 57 | PI_CNTR_INVALID_RECORDER_CHAN | Data record table does not exist |
| 58 | PI_CNTR_INVALID_RECORDER_SRC_OPT | Source does not exist; number too low or too high |
| 59 | PI_CNTR_INVALID_RECORDER_SRC_CHAN | Source record table number too low or too high |
| 60 | PI_CNTR_PARAM_PROTECTION | Protected Param: Current Command Level (CCL) too low |
| 61 | PI_CNTR_AUTOZERO_RUNNING | Command execution not possible while autozero is running |
| 62 | PI_CNTR_NO_LINEAR_AXIS | Autozero requires at least one linear axis |
| 63 | PI_CNTR_INIT_RUNNING | Initialization still in progress |
| 64 | PI_CNTR_READ_ONLY_PARAMETER | Parameter is read-only |
| 65 | PI_CNTR_PAM_NOT_FOUND | Parameter not found in nonvolatile memory |
| 66 | PI_CNTR_VOL_OUT_OF_LIMITS | Voltage out of limits |
| 67 | PI_CNTR_WAVE_TOO_LARGE | Not enough memory available for requested wave curve |
| 68 | PI_CNTR_NOT_ENOUGH_DDL_MEMORY | Not enough memory available for DDL table; DDL cannot be started |
| 69 | PI_CNTR_DDL_TIME_DELAY_TOO_LARGE | Time delay larger than DDL table; DDL cannot be started |
| 70 | PI_CNTR_DIFFERENT_ARRAY_LENGTH | The requested arrays have different lengths; query them separately |
| 71 | PI_CNTR_GEN_SINGLE_MODE_RESTART | Attempt to restart the generator while it is running in single step mode |
| 72 | PI_CNTR_ANALOG_TARGET_ACTIVE | Motion commands and wave generator activation are not allowed when analog target is active |

| | | |
|-----|--|---|
| 73 | PI_CNTR_WAVE_GENERATOR_ACTIVE | Motion commands are not allowed when wave generator is active |
| 74 | PI_CNTR_AUTOZERO_DISABLED | No sensor channel or no piezo channel connected to selected axis (sensor and piezo matrix) |
| 75 | PI_CNTR_NO_WAVE_SELECTED | Generator started (WGO) without having selected a wave table (WSL). |
| 76 | PI_CNTR_IF_BUFFER_OVERRUN | Interface buffer overrun and command couldn't be received correctly |
| 77 | PI_CNTR_NOT_ENOUGH_RECORDED_DATA | Data record table does not hold enough recorded data |
| 78 | PI_CNTR_TABLE_DEACTIVATED | Data record table is not configured for recording |
| 79 | PI_CNTR_OPENLOOP_VALUE_SET_WHEN_SERVO_ON | Open-loop commands (SVA, SVR) are not allowed when servo is on |
| 80 | PI_CNTR_RAM_ERROR | Hardware error affecting RAM |
| 81 | PI_CNTR_MACRO_UNKNOWN_COMMAND | Not macro command |
| 82 | PI_CNTR_MACRO_PC_ERROR | Macro counter out of range |
| 83 | PI_CNTR_JOYSTICK_ACTIVE | Joystick is active |
| 84 | PI_CNTR_MOTOR_IS_OFF | Motor is off |
| 85 | PI_CNTR_ONLY_IN_MACRO | Macro-only command |
| 86 | PI_CNTR_JOYSTICK_UNKNOWN_AXIS | Invalid joystick axis |
| 87 | PI_CNTR_JOYSTICK_UNKNOWN_ID | Joystick unknown |
| 88 | PI_CNTR_REF_MODE_IS_ON | Move without referenced stage |
| 89 | PI_CNTR_NOT_ALLOWED_IN_CURRENT_MOTION_MODE | Command not allowed in current motion mode |
| 90 | PI_CNTR_DIO_AND_TRACING_NOT_POSSIBLE | No tracing possible while digital IOs are used on this HW revision. Reconnect to switch operation mode. |
| 91 | PI_CNTR_COLLISION | Move not possible, would cause collision |
| 92 | PI_CNTR_SLAVE_NOT_FAST_ENOUGH | Stage is not capable of following the master. Check the gear ratio. |
| 93 | PI_CNTR_CMD_NOT_ALLOWED_WHILE_AXIS_IN_MOTION | This command is not allowed while the affected axis or its master is in motion. |
| 94 | PI_CNTR_OPEN_LOOP_JOYSTICK_ENABLED | Servo cannot be switched on when open-loop joystick control is enabled. |
| 95 | PI_CNTR_INVALID_SERVO_STATE_FOR_PARAMETER | This parameter cannot be changed in current servo mode. |
| 96 | PI_CNTR_UNKNOWN_STAGE_NAME | Unknown stage name |
| 97 | PI_CNTR_INVALID_VALUE_LENGTH | Invalid length of value (too much characters) |
| 98 | PI_CNTR_AUTOZERO_FAILED | AutoZero procedure was not successful |
| 100 | PI_LABVIEW_ERROR | PI LabVIEW driver reports error. See source control for details. |
| 200 | PI_CNTR_NO_AXIS | No stage connected to axis |
| 201 | PI_CNTR_NO_AXIS_PARAM_FILE | File with axis parameters not found |
| 202 | PI_CNTR_INVALID_AXIS_PARAM_FILE | Invalid axis parameter file |
| 203 | PI_CNTR_NO_AXIS_PARAM_BACKUP | Backup file with axis parameters not found |
| 204 | PI_CNTR_RESERVED_204 | PI internal error code 204 |
| 205 | PI_CNTR_SMO_WITH_SERVO_ON | SMO with servo on |
| 206 | PI_CNTR_UUDECODE_INCOMPLETE_HEADER | uudecode: incomplete header |
| 207 | PI_CNTR_UUDECODE_NOTHING_TO_DECODE | uudecode: nothing to decode |
| 208 | PI_CNTR_UUDECODE_ILLEGAL_FORMAT | uudecode: illegal UUE format |
| 209 | PI_CNTR_CRC32_ERROR | CRC32 error |

| | | |
|-----|---|---|
| 210 | PI_CNTR_ILLEGAL_FILENAME | Illegal file name (must be 8-0 format) |
| 211 | PI_CNTR_FILE_NOT_FOUND | File not found on controller |
| 212 | PI_CNTR_FILE_WRITE_ERROR | Error writing file on controller |
| 213 | PI_CNTR_DTR_HINDERS_VELOCITY_CHANGE | VEL command not allowed in DTR command mode |
| 214 | PI_CNTR_POSITION_UNKNOWN | Position calculations failed |
| 215 | PI_CNTR_CONN_POSSIBLY_BROKEN | The connection between controller and stage may be broken |
| 216 | PI_CNTR_ON_LIMIT_SWITCH | The connected stage has driven into a limit switch, some controllers need CLR to resume operation |
| 217 | PI_CNTR_UNEXPECTED_STRUT_STOP | Strut test command failed because of an unexpected strut stop |
| 218 | PI_CNTR_POSITION_BASED_ON_ESTIMATION | While MOV! is running position can only be estimated! |
| 219 | PI_CNTR_POSITION_BASED_ON_INTERPOLATION | Position was calculated during MOV motion |
| 230 | PI_CNTR_INVALID_HANDLE | Invalid handle |
| 231 | PI_CNTR_NO_BIOS_FOUND | No bios found |
| 232 | PI_CNTR_SAVE_SYS_CFG_FAILED | Save system configuration failed |
| 233 | PI_CNTR_LOAD_SYS_CFG_FAILED | Load system configuration failed |
| 301 | PI_CNTR_SEND_BUFFER_OVERFLOW | Send buffer overflow |
| 302 | PI_CNTR_VOLTAGE_OUT_OF_LIMITS | Voltage out of limits |
| 303 | PI_CNTR_OPEN_LOOP_MOTION_SET_WHEN_SERVO_ON | Open-loop motion attempted when servo ON |
| 304 | PI_CNTR_RECEIVING_BUFFER_OVERFLOW | Received command is too long |
| 305 | PI_CNTR_EEPROM_ERROR | Error while reading/writing EEPROM |
| 306 | PI_CNTR_I2C_ERROR | Error on I2C bus |
| 307 | PI_CNTR_RECEIVING_TIMEOUT | Timeout while receiving command |
| 308 | PI_CNTR_TIMEOUT | A lengthy operation has not finished in the expected time |
| 309 | PI_CNTR_MACRO_OUT_OF_SPACE | Insufficient space to store macro |
| 310 | PI_CNTR_EUI_OLDVERSION_CFGDATA | Configuration data has old version number |
| 311 | PI_CNTR_EUI_INVALID_CFGDATA | Invalid configuration data |
| 333 | PI_CNTR_HARDWARE_ERROR | Internal hardware error |
| 400 | PI_CNTR_WAV_INDEX_ERROR | Wave generator index error |
| 401 | PI_CNTR_WAV_NOT_DEFINED | Wave table not defined |
| 402 | PI_CNTR_WAV_TYPE_NOT_SUPPORTED | Wave type not supported |
| 403 | PI_CNTR_WAV_LENGTH_EXCEEDS_LIMIT | Wave length exceeds limit |
| 404 | PI_CNTR_WAV_PARAMETER_NR | Wave parameter number error |
| 405 | PI_CNTR_WAV_PARAMETER_OUT_OF_LIMIT | Wave parameter out of range |
| 406 | PI_CNTR_WGO_BIT_NOT_SUPPORTED | WGO command bit not supported |
| 500 | PI_CNTR_EMERGENCY_STOP_BUTTON_ACTIVATED | The \"red knob\" is still set and disables system |
| 501 | PI_CNTR_EMERGENCY_STOP_BUTTON_WAS_ACTIVATED | The \"red knob\" was activated and still disables system - reanimation required |
| 502 | PI_CNTR_REDUNDANCY_LIMIT_EXCEEDED | Position consistency check failed |
| 503 | PI_CNTR_COLLISION_SWITCH_ACTIVATED | Hardware collision sensor(s) are activated |
| 504 | PI_CNTR_FOLLOWING_ERROR | Strut following error occurred, e.g. caused by overload or encoder failure |
| 505 | PI_CNTR_SENSOR_SIGNAL_INVALID | One sensor signal is not valid |

| | | |
|-----|---|---|
| 506 | PI_CNTR_SERVO_LOOP_UNSTABLE | Servo loop was unstable due to wrong parameter setting and switched off to avoid damage. |
| 507 | PI_CNTR_LOST_SPI_SLAVE_CONNECTION | Digital connection to external spi slave device is lost |
| 530 | PI_CNTR_NODE_DOES_NOT_EXIST | A command refers to a node that does not exist |
| 531 | PI_CNTR_PARENT_NODE_DOES_NOT_EXIST | A command refers to a node that has no parent node |
| 532 | PI_CNTR_NODE_IN_USE | Attempt to delete a node that is in use |
| 533 | PI_CNTR_NODE_DEFINITION_IS_CYCLIC | Definition of a node is cyclic |
| 536 | PI_CNTR_HEXAPOD_IN_MOTION | Transformation cannot be defined as long as Hexapod is in motion |
| 537 | PI_CNTR_TRANSFORMATION_TYPE_NOT_SUPPORTED | Transformation node cannot be activated |
| 539 | PI_CNTR_NODE_PARENT_IDENTICAL_TO_CHILD | A node cannot be linked to itself |
| 540 | PI_CNTR_NODE_DEFINITION_INCONSISTENT | Node definition is erroneous or not complete (replace or delete it) |
| 542 | PI_CNTR_NODES_NOT_IN_SAME_CHAIN | The nodes are not part of the same chain |
| 543 | PI_CNTR_NODE_MEMORY_FULL | Unused nodes must be deleted before new nodes can be stored |
| 544 | PI_CNTR_PIVOT_POINT_FEATURE_NOT_SUPPORTED | With some transformations pivot point usage is not supported |
| 545 | PI_CNTR_SOFTLIMITS_INVALID | Soft limits invalid due to changes in coordinate system |
| 546 | PI_CNTR_CS_WRITE_PROTECTED | Coordinate system is write protected |
| 547 | PI_CNTR_CS_CONTENT_FROM_CONFIG_FILE | Coordinate system cannot be changed because its content is loaded from a configuration file |
| 548 | PI_CNTR_CS_CANNOT_BE_LINKED | Coordinate system may not be linked |
| 549 | PI_CNTR_KSB_CS_ROTATION_ONLY | A KSB-type coordinate system can only be rotated by multiples of 90 degrees |
| 551 | PI_CNTR_CS_DATA_CANNOT_BE_QUERIED | This query is not supported for this coordinate system type |
| 552 | PI_CNTR_CS_COMBINATION_DOES_NOT_EXIST | This combination of work and tool coordinate systems does not exist |
| 553 | PI_CNTR_CS_COMBINATION_INVALID | The combination must consist of one work and one tool coordinate system |
| 554 | PI_CNTR_CS_TYPE_DOES_NOT_EXIST | This coordinate system type does not exist |
| 555 | PI_CNTR_UNKNOWN_ERROR | BasMac: unknown controller error |
| 556 | PI_CNTR_CS_TYPE_NOT_ACTIVATED | No coordinate system of this type is activated |
| 557 | PI_CNTR_CS_NAME_INVALID | Name of coordinate system is invalid |
| 558 | PI_CNTR_CS_GENERAL_FILE_MISSING | File with stored CS systems is missing or erroneous |
| 559 | PI_CNTR_CS_LEVELING_FILE_MISSING | File with leveling CS is missing or erroneous |
| 601 | PI_CNTR_NOT_ENOUGH_MEMORY | Not enough memory |
| 602 | PI_CNTR_HW_VOLTAGE_ERROR | Hardware voltage error |
| 603 | PI_CNTR_HW_TEMPERATURE_ERROR | Hardware temperature out of range |
| 604 | PI_CNTR_POSITION_ERROR_TOO_HIGH | Position error of any axis in the system is too high |
| 606 | PI_CNTR_INPUT_OUT_OF_RANGE | Maximum value of input signal has been exceeded |
| 607 | PI_CNTR_NO_INTEGER | Value is not integer |
| 608 | PI_CNTR_FAST_ALIGNMENT_PROCESS_IS_NOT_RUNNING | Fast alignment process cannot be paused because it is not running |

| | | |
|------|--|---|
| 609 | PI_CNTR_FAST_ALIGNMENT_PROCESS_IS_NOT_PAUSED | Fast alignment process cannot be restarted/resumed because it is not paused |
| 650 | PI_CNTR_UNABLE_TO_SET_PARAM_WITH_SPA | Parameter could not be set with SPA - SEP needed? |
| 651 | PI_CNTR_PHASE_FINDING_ERROR | Phase finding error |
| 652 | PI_CNTR_SENSOR_SETUP_ERROR | Sensor setup error |
| 653 | PI_CNTR_SENSOR_COMM_ERROR | Sensor communication error |
| 654 | PI_CNTR_MOTOR_AMPLIFIER_ERROR | Motor amplifier error |
| 655 | PI_CNTR_OVER_CURR_PROTEC_TRIGGERED_BY_I2T | Overcurrent protection triggered by I2T-module |
| 656 | PI_CNTR_OVER_CURR_PROTEC_TRIGGERED_BY_AMP_MODULE | Overcurrent protection triggered by amplifier module |
| 657 | PI_CNTR_SAFETY_STOP_TRIGGERED | Safety stop triggered |
| 658 | PI_SENSOR_OFF | Sensor off? |
| 700 | PI_CNTR_COMMAND_NOT_ALLOWED_IN_EXTERNAL_MODE | Command not allowed in external mode |
| 710 | PI_CNTR_EXTERNAL_MODE_ERROR | External mode communication error |
| 715 | PI_CNTR_INVALID_MODE_OF_OPERATION | Invalid mode of operation |
| 716 | PI_CNTR_FIRMWARE_STOPPED_BY_CMD | Firmware stopped by command (#27) |
| 717 | PI_CNTR_EXTERNAL_MODE_DRIVER_MISSING | External mode driver missing |
| 718 | PI_CNTR_CONFIGURATION_FAILURE_EXTERNAL_MODE | Missing or incorrect configuration of external mode |
| 719 | PI_CNTR_EXTERNAL_MODE_CYCLETIME_INVALID | External mode cycletime invalid |
| 720 | PI_CNTR_BRAKE_ACTIVATED | Brake is activated |
| 1000 | PI_CNTR_TOO_MANY_NESTED_MACROS | Too many nested macros |
| 1001 | PI_CNTR_MACRO_ALREADY_DEFINED | Macro already defined |
| 1002 | PI_CNTR_NO_MACRO_RECORDING | Macro recording not activated |
| 1003 | PI_CNTR_INVALID_MAC_PARAM | Invalid parameter for MAC |
| 1004 | PI_CNTR_RESERVED_1004 | PI internal error code 1004 |
| 1005 | PI_CNTR_CONTROLLER_BUSY | Controller is busy with some lengthy operation (e.g. reference move, fast scan algorithm) |
| 1006 | PI_CNTR_INVALID_IDENTIFIER | Invalid identifier (invalid special characters, ...) |
| 1007 | PI_CNTR_UNKNOWN_VARIABLE_OR_ARGUMENT | Variable or argument not defined |
| 1008 | PI_CNTR_RUNNING_MACRO | Controller is (already) running a macro |
| 1009 | PI_CNTR_MACRO_INVALID_OPERATOR | Invalid or missing operator for condition. Check necessary spaces around operator. |
| 1010 | PI_CNTR_MACRO_NO_ANSWER | No answer was received while executing WAC/MEX/JRC/... |
| 1011 | PI_CMD_NOT_VALID_IN_MACRO_MODE | Command not valid during macro execution |
| 1024 | PI_CNTR_MOTION_ERROR | Motion error: position error too large, servo is switched off automatically |
| 1063 | PI_CNTR_EXT_PROFILE_UNALLOWED_CMD | User profile mode: command is not allowed, check for required preparatory commands |
| 1064 | PI_CNTR_EXT_PROFILE_EXPECTING_MOTION_ERROR | User profile mode: first target position in user profile is too far from current position |
| 1065 | PI_CNTR_PROFILE_ACTIVE | Controller is (already) in user profile mode |
| 1066 | PI_CNTR_PROFILE_INDEX_OUT_OF_RANGE | User profile mode: block or data set index out of allowed range |
| 1071 | PI_CNTR_PROFILE_OUT_OF_MEMORY | User profile mode: out of memory |
| 1072 | PI_CNTR_PROFILE_WRONG_CLUSTER | User profile mode: cluster is not assigned to this axis |
| 1073 | PI_CNTR_PROFILE_UNKNOWN_CLUSTER_IDENTIFIER | Unknown cluster identifier |

| | | |
|------|---------------------------------------|--|
| 1090 | PI_CNTR_TOO_MANY_TCP_CONNECTIONS_OPEN | There are too many open tcpip connections |
| 2000 | PI_CNTR_ALREADY_HAS_SERIAL_NUMBER | Controller already has a serial number |
| 4000 | PI_CNTR_SECTOR_ERASE_FAILED | Sector erase failed |
| 4001 | PI_CNTR_FLASH_PROGRAM_FAILED | Flash program failed |
| 4002 | PI_CNTR_FLASH_READ_FAILED | Flash read failed |
| 4003 | PI_CNTR_HW_MATCHCODE_ERROR | HW match code missing/invalid |
| 4004 | PI_CNTR_FW_MATCHCODE_ERROR | FW match code missing/invalid |
| 4005 | PI_CNTR_HW_VERSION_ERROR | HW version missing/invalid |
| 4006 | PI_CNTR_FW_VERSION_ERROR | FW version missing/invalid |
| 4007 | PI_CNTR_FW_UPDATE_ERROR | FW update failed |
| 4008 | PI_CNTR_FW_CRC_PAR_ERROR | FW Parameter CRC wrong |
| 4009 | PI_CNTR_FW_CRC_FW_ERROR | FW CRC wrong |
| 5000 | PI_CNTR_INVALID_PCC_SCAN_DATA | PicoCompensation scan data is not valid |
| 5001 | PI_CNTR_PCC_SCAN_RUNNING | PicoCompensation is running, some actions cannot be executed during scanning/recording |
| 5002 | PI_CNTR_INVALID_PCC_AXIS | Given axis cannot be defined as PPC axis |
| 5003 | PI_CNTR_PCC_SCAN_OUT_OF_RANGE | Defined scan area is larger than the travel range |
| 5004 | PI_CNTR_PCC_TYPE_NOT_EXISTING | Given PicoCompensation type is not defined |
| 5005 | PI_CNTR_PCC_PAM_ERROR | PicoCompensation parameter error |
| 5006 | PI_CNTR_PCC_TABLE_ARRAY_TOO_LARGE | PicoCompensation table is larger than maximum table length |
| 5100 | PI_CNTR_NEXLINE_ERROR | Common error in NEXLINE® firmware module |
| 5101 | PI_CNTR_CHANNEL_ALREADY_USED | Output channel for NEXLINE® cannot be redefined for other usage |
| 5102 | PI_CNTR_NEXLINE_TABLE_TOO_SMALL | Memory for NEXLINE® signals is too small |
| 5103 | PI_CNTR_RNP_WITH_SERVO_ON | RNP cannot be executed if axis is in closed loop |
| 5104 | PI_CNTR_RNP_NEEDED | Relax procedure (RNP) needed |
| 5200 | PI_CNTR_AXIS_NOT_CONFIGURED | Axis must be configured for this action |
| 5300 | PI_CNTR_FREQU_ANALYSIS_FAILED | Frequency analysis failed |
| 5301 | PI_CNTR_FREQU_ANALYSIS_RUNNING | Another frequency analysis is running |
| 6000 | PI_CNTR_SENSOR_ABS_INVALID_VALUE | Invalid preset value of absolute sensor |
| 6001 | PI_CNTR_SENSOR_ABS_WRITE_ERROR | Error while writing to sensor |
| 6002 | PI_CNTR_SENSOR_ABS_READ_ERROR | Error while reading from sensor |
| 6003 | PI_CNTR_SENSOR_ABS_CRC_ERROR | Checksum error of absolute sensor |
| 6004 | PI_CNTR_SENSOR_ABS_ERROR | General error of absolute sensor |
| 6005 | PI_CNTR_SENSOR_ABS_OVERFLOW | Overflow of absolute sensor position |

Interface errors

| | | |
|----|---------------------|---|
| 0 | COM_NO_ERROR | No error occurred during function call |
| -1 | COM_ERROR | Error during com operation (could not be specified) |
| -2 | SEND_ERROR | Error while sending data |
| -3 | REC_ERROR | Error while receiving data |
| -4 | NOT_CONNECTED_ERROR | Not connected (no port with given ID open) |
| -5 | COM_BUFFER_OVERFLOW | Buffer overflow |

| | | |
|-----|---|---|
| -6 | CONNECTION_FAILED | Error while opening port |
| -7 | COM_TIMEOUT | Timeout error |
| -8 | COM_MULTILINE_RESPONSE | There are more lines waiting in buffer |
| -9 | COM_INVALID_ID | There is no interface or DLL handle with the given ID |
| -10 | COM_NOTIFY_EVENT_ERROR | Event/message for notification could not be opened |
| -11 | COM_NOT_IMPLEMENTED | Function not supported by this interface type |
| -12 | COM_ECHO_ERROR | Error while sending "echoed" data |
| -13 | COM_GPIB_EDVR | IEEE488: System error |
| -14 | COM_GPIB_ECIC | IEEE488: Function requires GPIB board to be CIC |
| -15 | COM_GPIB_ENOL | IEEE488: Write function detected no listeners |
| -16 | COM_GPIB_EADR | IEEE488: Interface board not addressed correctly |
| -17 | COM_GPIB_EARG | IEEE488: Invalid argument to function call |
| -18 | COM_GPIB_ESAC | IEEE488: Function requires GPIB board to be SAC |
| -19 | COM_GPIB_EABO | IEEE488: I/O operation aborted |
| -20 | COM_GPIB_ENEB | IEEE488: Interface board not found |
| -21 | COM_GPIB_EDMA | IEEE488: Error performing DMA |
| -22 | COM_GPIB_EOIP | IEEE488: I/O operation started before previous operation completed |
| -23 | COM_GPIB_ECAP | IEEE488: No capability for intended operation |
| -24 | COM_GPIB_EFSO | IEEE488: File system operation error |
| -25 | COM_GPIB_EBUS | IEEE488: Command error during device call |
| -26 | COM_GPIB_ESTB | IEEE488: Serial poll-status byte lost |
| -27 | COM_GPIB_ESRQ | IEEE488: SRQ remains asserted |
| -28 | COM_GPIB_ETAB | IEEE488: Return buffer full |
| -29 | COM_GPIB_ELCK | IEEE488: Address or board locked |
| -30 | COM_RS_INVALID_DATA_BITS | RS-232: 5 data bits with 2 stop bits is an invalid combination, as is 6, 7, or 8 data bits with 1.5 stop bits |
| -31 | COM_ERROR_RS_SETTINGS | RS-232: Error configuring the COM port |
| -32 | COM_INTERNAL_RESOURCES_ERROR | Error dealing with internal system resources (events, threads, ...) |
| -33 | COM_DLL_FUNC_ERROR | A DLL or one of the required functions could not be loaded |
| -34 | COM_FTDIUSB_INVALID_HANDLE | FTDIUSB: invalid handle |
| -35 | COM_FTDIUSB_DEVICE_NOT_FOUND | FTDIUSB: device not found |
| -36 | COM_FTDIUSB_DEVICE_NOT_OPENED | FTDIUSB: device not opened |
| -37 | COM_FTDIUSB_IO_ERROR | FTDIUSB: IO error |
| -38 | COM_FTDIUSB_INSUFFICIENT_RESOURCES | FTDIUSB: insufficient resources |
| -39 | COM_FTDIUSB_INVALID_PARAMETER | FTDIUSB: invalid parameter |
| -40 | COM_FTDIUSB_INVALID_BAUD_RATE | FTDIUSB: invalid baud rate |
| -41 | COM_FTDIUSB_DEVICE_NOT_OPENED_FOR_ERASE | FTDIUSB: device not opened for erase |
| -42 | COM_FTDIUSB_DEVICE_NOT_OPENED_FOR_WRITE | FTDIUSB: device not opened for write |
| -43 | COM_FTDIUSB_FAILED_TO_WRITE_DEVICE | FTDIUSB: failed to write device |
| -44 | COM_FTDIUSB_EEPROM_READ_FAILED | FTDIUSB: EEPROM read failed |
| -45 | COM_FTDIUSB_EEPROM_WRITE_FAILED | FTDIUSB: EEPROM write failed |
| -46 | COM_FTDIUSB_EEPROM_ERASE_FAILED | FTDIUSB: EEPROM erase failed |
| -47 | COM_FTDIUSB_EEPROM_NOT_PRESENT | FTDIUSB: EEPROM not present |
| -48 | COM_FTDIUSB_EEPROM_NOT_PROGRAMMED | FTDIUSB: EEPROM not programmed |
| -49 | COM_FTDIUSB_INVALID_ARGS | FTDIUSB: invalid arguments |

| | | |
|-----|---------------------------|--|
| -50 | COM_FTDIUSB_NOT_SUPPORTED | FTDIUSB: not supported |
| -51 | COM_FTDIUSB_OTHER_ERROR | FTDIUSB: other error |
| -52 | COM_PORT_ALREADY_OPEN | Error while opening the COM port: was already open |
| -53 | COM_PORT_CHECKSUM_ERROR | Checksum error in received data from COM port |
| -54 | COM_SOCKET_NOT_READY | Socket not ready, you should call the function again |
| -55 | COM_SOCKET_PORT_IN_USE | Port is used by another socket |
| -56 | COM_SOCKET_NOT_CONNECTED | Socket not connected (or not valid) |
| -57 | COM_SOCKET_TERMINATED | Connection terminated (by peer) |
| -58 | COM_SOCKET_NO_RESPONSE | Can't connect to peer |
| -59 | COM_SOCKET_INTERRUPTED | Operation was interrupted by a nonblocked signal |
| -60 | COM_PCI_INVALID_ID | No device with this ID is present |
| -61 | COM_PCI_ACCESS_DENIED | Driver could not be opened (on Vista: run as administrator!) |
| -62 | COM_SOCKET_HOST_NOT_FOUND | Host not found |
| -63 | COM_DEVICE_CONNECTED | Device already connected |

DLL errors

| | | |
|-------|-----------------------------|--|
| -1001 | PI_UNKNOWN_AXIS_IDENTIFIER | Unknown axis identifier |
| -1002 | PI_NR_NAV_OUT_OF_RANGE | Number for NAV out of range--must be in [1,10000] |
| -1003 | PI_INVALID_SGA | Invalid value for SGA--must be one of 1, 10, 100, 1000 |
| -1004 | PI_UNEXPECTED_RESPONSE | Controller sent unexpected response |
| -1005 | PI_NO_MANUAL_PAD | No manual control pad installed, calls to SMA and related commands are not allowed |
| -1006 | PI_INVALID_MANUAL_PAD_KNOB | Invalid number for manual control pad knob |
| -1007 | PI_INVALID_MANUAL_PAD_AXIS | Axis not currently controlled by a manual control pad |
| -1008 | PI_CONTROLLER_BUSY | Controller is busy with some lengthy operation (e.g. reference move, fast scan algorithm) |
| -1009 | PI_THREAD_ERROR | Internal error--could not start thread |
| -1010 | PI_IN_MACRO_MODE | Controller is (already) in macro mode--command not valid in macro mode |
| -1011 | PI_NOT_IN_MACRO_MODE | Controller not in macro mode--command not valid unless macro mode active |
| -1012 | PI_MACRO_FILE_ERROR | Could not open file to write or read macro |
| -1013 | PI_NO_MACRO_OR_EMPTY | No macro with given name on controller, or macro is empty |
| -1014 | PI_MACRO_EDITOR_ERROR | Internal error in macro editor |
| -1015 | PI_INVALID_ARGUMENT | One or more arguments given to function is invalid (empty string, index out of range, ...) |
| -1016 | PI_AXIS_ALREADY_EXISTS | Axis identifier is already in use by a connected stage |
| -1017 | PI_INVALID_AXIS_IDENTIFIER | Invalid axis identifier |
| -1018 | PI_COM_ARRAY_ERROR | Could not access array data in COM server |
| -1019 | PI_COM_ARRAY_RANGE_ERROR | Range of array does not fit the number of parameters |
| -1020 | PI_INVALID_SPA_CMD_ID | Invalid parameter ID given to SPA or SPA? |
| -1021 | PI_NR_AVG_OUT_OF_RANGE | Number for AVG out of range--must be >0 |
| -1022 | PI_WAV_SAMPLES_OUT_OF_RANGE | Incorrect number of samples given to WAV |
| -1023 | PI_WAV_FAILED | Generation of wave failed |
| -1024 | PI_MOTION_ERROR | Motion error: position error too large, servo is switched off automatically |
| -1025 | PI_RUNNING_MACRO | Controller is (already) running a macro |

| | | |
|-------|---|---|
| -1026 | PI_PZT_CONFIG_FAILED | Configuration of PZT stage or amplifier failed |
| -1027 | PI_PZT_CONFIG_INVALID_PARAMS | Current settings are not valid for desired configuration |
| -1028 | PI_UNKNOWN_CHANNEL_IDENTIFIER | Unknown channel identifier |
| -1029 | PI_WAVE_PARAM_FILE_ERROR | Error while reading/writing wave generator parameter file |
| -1030 | PI_UNKNOWN_WAVE_SET | Could not find description of wave form. Maybe WG.INI is missing? |
| -1031 | PI_WAVE_EDITOR_FUNC_NOT_LOADED | The WGWaveEditor DLL function was not found at startup |
| -1032 | PI_USER_CANCELLED | The user cancelled a dialog |
| -1033 | PI_C844_ERROR | Error from C-844 Controller |
| -1034 | PI_DLL_NOT_LOADED | DLL necessary to call function not loaded, or function not found in DLL |
| -1035 | PI_PARAMETER_FILE_PROTECTED | The open parameter file is protected and cannot be edited |
| -1036 | PI_NO_PARAMETER_FILE_OPENED | There is no parameter file open |
| -1037 | PI_STAGE_DOES_NOT_EXIST | Selected stage does not exist |
| -1038 | PI_PARAMETER_FILE_ALREADY_OPENED | There is already a parameter file open. Close it before opening a new file |
| -1039 | PI_PARAMETER_FILE_OPEN_ERROR | Could not open parameter file |
| -1040 | PI_INVALID_CONTROLLER_VERSION | The version of the connected controller is invalid |
| -1041 | PI_PARAM_SET_ERROR | Parameter could not be set with SPA--parameter not defined for this controller! |
| -1042 | PI_NUMBER_OF_POSSIBLE_WAVES_EXCEEDED | The maximum number of wave definitions has been exceeded |
| -1043 | PI_NUMBER_OF_POSSIBLE_GENERATORS_EXCEEDED | The maximum number of wave generators has been exceeded |
| -1044 | PI_NO_WAVE_FOR_AXIS_DEFINED | No wave defined for specified axis |
| -1045 | PI_CANT_STOP_OR_START_WAV | Wave output to axis already stopped/started |
| -1046 | PI_REFERENCE_ERROR | Not all axes could be referenced |
| -1047 | PI_REQUIRED_WAVE_NOT_FOUND | Could not find parameter set required by frequency relation |
| -1048 | PI_INVALID_SPP_CMD_ID | Command ID given to SPP or SPP? is not valid |
| -1049 | PI_STAGE_NAME_ISNT_UNIQUE | A stage name given to CST is not unique |
| -1050 | PI_FILE_TRANSFER_BEGIN_MISSING | A uuencoded file transferred did not start with "begin" followed by the proper filename |
| -1051 | PI_FILE_TRANSFER_ERROR_TEMP_FILE | Could not create/read file on host PC |
| -1052 | PI_FILE_TRANSFER_CRC_ERROR | Checksum error when transferring a file to/from the controller |
| -1053 | PI_COULDNT_FIND_PISTAGES_DAT | The PiStages.dat database could not be found. This file is required to connect a stage with the CST command |
| -1054 | PI_NO_WAVE_RUNNING | No wave being output to specified axis |
| -1055 | PI_INVALID_PASSWORD | Invalid password |
| -1056 | PI_OPM_COM_ERROR | Error during communication with OPM (Optical Power Meter), maybe no OPM connected |
| -1057 | PI_WAVE_EDITOR_WRONG_PARAMNUM | WaveEditor: Error during wave creation, incorrect number of parameters |
| -1058 | PI_WAVE_EDITOR_FREQUENCY_OUT_OF_RANGE | WaveEditor: Frequency out of range |
| -1059 | PI_WAVE_EDITOR_WRONG_IP_VALUE | WaveEditor: Error during wave creation, incorrect index for integer parameter |
| -1060 | PI_WAVE_EDITOR_WRONG_DP_VALUE | WaveEditor: Error during wave creation, incorrect index for floating point parameter |
| -1061 | PI_WAVE_EDITOR_WRONG_ITEM_VALUE | WaveEditor: Error during wave creation, could not calculate value |

| | | |
|-------|--|---|
| -1062 | PI_WAVE_EDITOR_MISSING_GRAPH_COMPONENT | WaveEditor: Graph display component not installed |
| -1063 | PI_EXT_PROFILE_UNALLOWED_CMD | User Profile Mode: Command is not allowed, check for required preparatory commands |
| -1064 | PI_EXT_PROFILE_EXPECTING_MOTION_ERROR | User Profile Mode: First target position in User Profile is too far from current position |
| -1065 | PI_EXT_PROFILE_ACTIVE | Controller is (already) in User Profile Mode |
| -1066 | PI_EXT_PROFILE_INDEX_OUT_OF_RANGE | User Profile Mode: Block or Data Set index out of allowed range |
| -1067 | PI_PROFILE_GENERATOR_NO_PROFILE | ProfileGenerator: No profile has been created yet |
| -1068 | PI_PROFILE_GENERATOR_OUT_OF_LIMITS | ProfileGenerator: Generated profile exceeds limits of one or both axes |
| -1069 | PI_PROFILE_GENERATOR_UNKNOWN_PARAMETER | ProfileGenerator: Unknown parameter ID in Set/Get Parameter command |
| -1070 | PI_PROFILE_GENERATOR_PAR_OUT_OF_RANGE | ProfileGenerator: Parameter out of allowed range |
| -1071 | PI_EXT_PROFILE_OUT_OF_MEMORY | User Profile Mode: Out of memory |
| -1072 | PI_EXT_PROFILE_WRONG_CLUSTER | User Profile Mode: Cluster is not assigned to this axis |
| -1073 | PI_UNKNOWN_CLUSTER_IDENTIFIER | Unknown cluster identifier |
| -1074 | PI_INVALID_DEVICE_DRIVER_VERSION | The installed device driver doesn't match the required version. Please see the documentation to determine the required device driver version. |
| -1075 | PI_INVALID_LIBRARY_VERSION | The library used doesn't match the required version. Please see the documentation to determine the required library version. |
| -1076 | PI_INTERFACE_LOCKED | The interface is currently locked by another function. Please try again later. |
| -1077 | PI_PARAM_DAT_FILE_INVALID_VERSION | Version of parameter DAT file does not match the required version. Current files are available at www.pi.ws . |
| -1078 | PI_CANNOT_WRITE_TO_PARAM_DAT_FILE | Cannot write to parameter DAT file to store user defined stage type. |
| -1079 | PI_CANNOT_CREATE_PARAM_DAT_FILE | Cannot create parameter DAT file to store user defined stage type. |
| -1080 | PI_PARAM_DAT_FILE_INVALID_REVISION | Parameter DAT file does not have correct revision. |
| -1081 | PI_USERSTAGES_DAT_FILE_INVALID_REVISION | User stages DAT file does not have correct revision. |
| -1082 | PI_SOFTWARE_TIMEOUT | Timeout Error. Some lengthy operation did not finish within expected time. |
| -1083 | PI_WRONG_DATA_TYPE | A function argument has an unexpected data type. |
| -1084 | PI_DIFFERENT_ARRAY_SIZES | Length of data arrays is different. |
| -1085 | PI_PARAM_NOT_FOUND_IN_PARAM_DAT_FILE | Parameter value not found in parameter DAT file. |
| -1086 | PI_MACRO_RECORDING_NOT_ALLOWED_IN_THIS_MODE | Macro recording is not allowed in this mode of operation. |
| -1087 | PI_USER_CANCELLED_COMMAND | Command cancelled by user input. |
| -1088 | PI_TOO_FEW_GCS_DATA | Controller sent too few GCS data sets |
| -1089 | PI_TOO_MANY_GCS_DATA | Controller sent too many GCS data sets |
| -1090 | PI_GCS_DATA_READ_ERROR | Communication error while reading GCS data |
| -1091 | PI_WRONG_NUMBER_OF_INPUT_ARGUMENTS | Wrong number of input arguments |
| -1092 | PI_FAILED_TO_CHANGE_CCL_LEVEL | Change of command level has failed |
| -1093 | PI_FAILED_TO_SWITCH_OFF_SERVO | Switching off the servo mode has failed |
| -1094 | PI_FAILED_TO_SET_SINGLE_PARAMETER_WHILE_PERFORMING_CST | A parameter could not be set while performing CST: CST was not performed (parameters remain unchanged). |
| -1095 | PI_ERROR_CONTROLLER_REBOOT | Connection could not be reestablished after reboot. |
| -1096 | PI_ERROR_AT_QHPA | Sending HPA? or receiving the response has failed. |
| -1097 | PI_QHPA_NONCOMPLIANT_WITH_GCS | HPA? response does not comply with GCS2 syntax. |

| | | |
|-------|----------------------------|--|
| -1098 | PI_FAILED_TO_READ_QSPA | Response to SPA? could not be received. |
| -1099 | PI_PAM_FILE_WRONG_VERSION | Version of PAM file cannot be handled (too old or too new) |
| -1100 | PI_PAM_FILE_INVALID_FORMAT | PAM file does not contain required data in PAM-file format |
| -1101 | PI_INCOMPLETE_INFORMATION | Information does not contain all required data |
| -1102 | PI_NO_VALUE_AVAILABLE | No value for parameter available |
| -1103 | PI_NO_PAM_FILE_OPEN | No PAM file is open |
| -1104 | PI_INVALID_VALUE | Invalid value |
| -1105 | PI_UNKNOWN_PARAMETER | Unknown parameter |

