

SM152E Software Manual

E-861 Windows GCS 2.0 DLL (PI_GCS2_DLL)

Release: 1.1.1

Date: 2010-07-26



This document describes software for use with the following product(s):

- E-861.1A1
NEXACT® Controller, 1 channel, linear encoder

Physik Instrumente (PI) GmbH & Co. KG is the owner of the following company names and trademarks:
PI®, PIC®, PICMA®, PILine®, PIFOC®, PiezoWalk®, NEXACT®, NEXLINE®, NanoCube®, NanoAutomation®

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

The products described in this document are in part protected by the following patents:
German Patent No. P4408618.0

Copyright by 1999–2010 Physik Instrumente (PI) GmbH & Co. KG, Karlsruhe, Germany
The text, photographs and drawings in this manual enjoy copyright protection. With regard thereto, Physik Instrumente (PI) GmbH & Co. KG reserves all rights. Use of said text, photographs and drawings is permitted only in part and only upon citation of the source.

First printing 26 July 2010
Document Number SM152E, BRo, KSch, Release 1.1.1
E-861_PIGCS_2_0_DLL_SM152E

Subject to change without notice. This manual is superseded by any new release. The newest release is available for download at www.pi.ws (<http://www.pi.ws>).

Table of Contents

0.	Disclaimer	2
1.	Introduction.....	3
1.1.	Accessible Items and their Identifiers	3
1.2.	Conversion of Units.....	4
1.3.	Rounding Considerations.....	4
2.	Quick Start	5
2.1.	Software Installation.....	5
2.2.	Connect the Controller	5
2.3.	Install USB Drivers	5
2.4.	Starting Up	6
2.5.	Referencing (Closed-Loop Systems Only).....	7
2.6.	Application Notes	7
2.7.	Samples	8
2.7.1.	Stand-Alone E-861 System.....	8
2.7.2.	E-861 System(s) in a Daisy Chain	9
3.	General Information About PI DLLs	10
3.1.	Threads.....	10
3.2.	DLL Handling	10
3.2.1.	Using a Static Import Library.....	10
3.2.2.	Using a Module Definition File	11
3.2.3.	Using Windows API Functions.....	11
3.3.	Function Calls	11
3.3.1.	Error Return	11
3.3.2.	Axis Identifiers.....	12
3.3.3.	Axis Parameters.....	12
3.4.	Types Used in PI Software	12
3.4.1.	Boolean Values.....	12
3.4.2.	NULL Pointers.....	12
3.4.3.	C-Strings	12
4.	Communication Functions.....	13
4.1.	Usage and Overview.....	13
4.2.	Function Description	14
4.3.	Interface Settings	17
5.	Functions for Sending and Reading Strings.....	18

5.1.	Overview	19
5.2.	Function Description	19
6.	Basic Functions for GCS Commands	20
6.1.	Overview	20
6.2.	Function Description	25
7.	Functions for User-Defined Stages	59
7.1.	Overview	59
7.2.	Function Description	60
7.3.	Parameter Databases	61
7.4.	Updating PISTages2.dat	62
7.5.	Troubleshooting	62
8.	System Parameter Overview	64
9.	Error Codes	71
	Controller Errors	71
10.	Index	86

0. Disclaimer

This software is provided "as is". PI does not guarantee that this software is free of errors and will not be responsible for any damage arising from the use of this software. The user agrees to use this software on his own responsibility.

1. Introduction

The PI_GCS2_DLL allows controlling one or more PI E-861 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** XP (32 bit), Vista (32 bit, 64 bit) and Windows 7 (32 bit, 64 bit): PI_GCS2_DLL
See Section 3.2 starting on p. 10 for more information about PI DLLs.

1.1. Accessible Items and their Identifiers

The identifiers listed below are used to address the appropriate items with the commands of the PI General Command Set (GCS) which is supported by the firmware of the E-861. Note that you can address only one single item (e.g. axis or channel) per function call, or, with queries and if the function supports this, address all items by omitting the item identifier.

The identifiers of the following items are factory defaults and cannot be changed by the user:

- **Logical axis/PiezoWalk channel:** one axis/PiezoWalk channel, the identifier is 1.
In the E-861 firmware, motion for logical axes (i.e. for the directions of motion of a stage) is commanded by closed-loop move functions PI_MOV() and PI_MVR() and with the open-loop move functions PI_OMA() and PI_OMR().
Motion for PiezoWalk channels (i.e. for single NEXACT® linear drives) is commanded using move functions like PI_OAD() or PI_OSM().
Since the E-861 is a single-axis / single-channel device, the terms "axis" and "PiezoWalk channel" can be used synonymously.
- **Analog input channels:** six channels, the identifiers are 1 to 6.
"Genuine" analog input lines, with the identifiers 1 to 4, are Input 1 to Input 4 on the I/O socket. Their number is reported by PI_qTAC(), and their values can be queried with PI_qTAV(). Note that these lines can also be used for digital input (see below).
Further analog input lines are located on the Joystick socket: channel 5 is the input line for the joystick axis and 6 the input line for the joystick button. They are not reported by PI_qTAC() and PI_qTAV(). See also the Joystick information below.
The values of all six channels can be recorded using the record option 81 of PI_DRC().
- **Digital output lines:** four lines, the identifiers are 1 to 4.
1 to 4 identify the Output 1 to Output 4 digital output lines on the I/O socket.
See "Using Trigger Input and Output" in the E-861 User manual for more information.
- **Digital input lines:** four lines, the identifiers are 1 to 4.
1 to 4 identify the Input 1 to Input 4 digital input lines on the I/O socket which can also be used for analog input (see above).
See "Using Trigger Input and Output" in the E-861 User manual for more information.
- **Joystick:** one joystick device, identifier is 1 with all joystick-related commands. Note that the second joystick device shown in some responses is an analog input which is currently deactivated and provided for future applications.
The E-861 supports one axis and one button of the joystick.
The identifier of the joystick axis is 1 with joystick-related functions (PI_qJAS(), PI_JAX(), PI_qJAX(), PI_JDT()) and 5 with PI_DRC(), record option 81.
The identifier of the joystick button is 1 with the joystick-related PI_qJBS() function and 6 with

PI_DRC(), record option 81.

See "Joystick Control" and "Joystick Socket" in the E-861 User manual for more information.

- **Data recorder tables (memory tables for recorded data):** 2 tables with 1024 points per table, the identifiers are 1 and 2

See "Data Recording" in the E-861 User manual for more information.

Each E-861 must have a unique controller address. The controller address can be changed by the user if, for example, a daisy chain network is to be set up:

- **Controller address:** the E-861 device address in the range of 1 to 16, can be set with the DIP switches on the front panel, see "DIP Switch Settings" and "Target and Sender Address" in the E-861 User manual for details.

1.2. Conversion of Units

The GCS system uses physical units of measure. Most controllers and GCS software have default conversion factors chosen to convert hardware-dependent units (e.g. encoder counts) into mm or degrees, as appropriate (see PI_SPA () and PI_qSPA (), parameters 0xE and 0xF). These defaults are generally taken from a database of stages that can be connected.

1.3. Rounding Considerations

When converting commanded position values from physical units to the hardware-dependent units required by the motion control layers, rounding errors can occur. The GCS software is so designed, that a relative move of x working units will always result in a relative move of the same number of hardware units. Because of rounding errors, this means, for example, that 2 relative moves of x working units may differ slightly from one relative move of $2x$. When making large numbers of relative moves, especially when moving back and forth, either intersperse absolute moves, or make sure that each relative move in one direction is matched by a relative move of the same size in the other direction.

Examples:

Assuming 5 hardware units = 33×10^{-6} working units:

Relative moves smaller than 0.000003 working units cause move of 0 hardware units.

Relative moves of 0.000004 to 0.000009 working units cause move of 1 hardware unit.

Relative moves of 0.000010 to 0.000016 working units cause move of 2 hardware units.

Relative moves of 0.000017 to 0.000023 working units cause move of 3 hardware units.

Relative moves of 0.000024 to 0.000029 working units cause move of 4 hardware units.

Hence:

2 moves of 10×10^{-6} working units followed by 1 move of 20×10^{-6} in the other direction cause a net motion of 1 hardware unit forward.

100 moves of 22×10^{-6} followed by 200 of -11×10^{-6} result in a net motion of -100 hardware units.


5000 moves of 2×10^{-6} result in no motion.

2. Quick Start

2.1. Software Installation

To install the E-861 PI GCS 2 DLL on your host PC, proceed as follows:

Windows operating systems:

- 1 Insert the product CD in your host PC.
- 2 If the Setup Wizard does not open automatically, start it from the root directory of the CD with the  icon.
- 3 Follow the on-screen instructions and select the “typical” installation. Typical components are LabVIEW drivers, GCS DLL, PIMikroMove.

NOTE

The PISTages2.dat stage database file is 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:\\Documents and Settings\\All Users\\Application Data (Windows XP) or C:\\ProgramData (Windows Vista). 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* entry in the controller menu or the *Search for controller software* entry in the *Connections* menu to identify the GCSTranslator path.

See Section 3 starting on p. 10 for more information about PI DLLs.

The PI host software is improved continually. It is therefore recommended that you visit the PI website (www.pi.ws) regularly to see if updated releases of the software are available for download. Updates are accompanied by information (readme files) so that you can decide if updating makes sense for your application.

You need a password to see if updates are available and to download them. This password is provided on the product CD in the *Produktname* ReleaseNews PDF file in the \\Manuals directory. See "Software Updates" in the User Manual of your controller for download details.

2.2. Connect the Controller

Physically connect the controller to the PC. Never connect both USB and RS-232 cables to the host at the same time. See the controller User Manual for details.

2.3. Install USB Drivers

When the USB interface to the controller is connected for the first time, you will be given the opportunity to install the drivers; this may be done at any time, though admin rights are required. Choose to select the device from a list, and give the “\\USB_Drivers” directory on the product CD as the location to search.

2.4. Starting Up

CAUTION

If no sensor is present in your system, do not switch the servo on with `PI_SVO()` and do not call functions that require a sensor, like `PI_MOV()`, `PI_MVR()`, `PI_OMA()` or `PI_OMR()`. Otherwise the connected mechanics can run into the hardstop at full speed which may cause damage to your hardware setup.

After all required files have been installed, write a program that performs the following steps:

- Open a connection between the host PC and the E-861, e.g. by calling `PI_ConnectRS232()`.

NOTE

When establishing a connection between the host PC and an E-861 the controller must have address 1.

If a daisy chain is to be connected to the host PC use the `PI_OpenRS232DaisyChain()` or the `PI_OpenUSBDAisyChain()` function **and** `PI_ConnectDaisyChainDevice()` function.

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.

See p. 13 for details.

- With open-loop systems (i.e. no sensor is present):
Command open-loop motion of the NEXACT® linear drive using a command sequence like the one listed below. See "Modes of Operation" in the E-861 User manual and "Application Notes" on p. 7 for more information regarding the motion modes of a NEXACT® linear drive and the transitions between them.

Note that open-loop move functions `PI_OAD()` and `PI_OSM()` command PiezoWalk channels, whereas `PI_OMA()` and `PI_OMR()` command axes. See "Accessible Items and Their Identifiers" on p. 3 for more information).

In the example listed below an N-310.11 NEXACT® linear drive is connected to the E-861.

OAD 1 40	Open-loop analog motion of PiezoWalk channel 1, set feed voltage to +40 volts which corresponds to a motion of approx. 3.6 µm in positive direction
OAD 1 0	Open-loop analog motion of PiezoWalk channel 1, set feed voltage to 0 volts, so that the drive should move back to the starting position
RNP 1 0	"Relaxing", brings the PiezoWalk channel 1 to a full-holding-force, zero-drive-voltage "Relaxed" state, is required before the motion mode can be changed from analog to nanostepping motion in open-loop operation
OSM 1 100	Open-loop nanostepping motion, PiezoWalk channel 1 moves 100 steps in positive direction

OSM 1 -100	Open-loop nanostepping motion, PiezoWalk channel 1 moves 100 steps in negative direction
RNP 1 0	"Relaxing"
OAD 1 -55	Analog motion of approx. 5 μm in negative direction

- With closed-loop systems (incremental sensor is present):
Before closed-loop operation is possible, several controller parameters must be adjusted to match the properties of the mechanics and the sensor used for position feedback. See "Parameters for Customizing" and "Controller Parameters" in the E-861 User manual for more information.
Switch the servo on with PI_SVO(). The stage must be referenced before you can make absolute moves with functions like PI_MOV(). By default, referencing must be done using PI_FRF(), PI_FNL() or PI_FPL(), depending on the connected mechanics. A sensor must be present to use these functions.
Use PI_IsControllerReady() in a loop to observe completion of the referencing procedure, see also Section 2.5. Afterwards, make a few test moves with PI_MOV() so that you can verify your program's operation.
See also "Example for Commanding Motion" for closed-loop systems in the E-861 User manual.

2.5. Referencing (Closed-Loop Systems Only)

Upon startup or reboot, the controller has no way of knowing the absolute position of the connected axis. The axis is said to be "unreferenced" and no moves can be made. Moves can be made allowable in the following ways:

- The axis can be referenced. This involves moving it until it tips a reference or limit switch. See the PI_FRF(), PI_FNL() and PI_FPL() functions for details.
- The controller can be told to set the reference mode for the axis OFF and allow relative moves only, without knowledge of the absolute position. See the PI_RON() function for details.
- For axes with reference mode OFF, the controller can be told to assume the absolute position has a given value. See the PI_POS() function for details.

With PI_qFRF() you can query the current reference state of an axis (referenced or not).

See also the controller User manual.

2.6. Application Notes

After switching from closed-loop to open-loop operation using PI_SVO(), the open-loop motion functions (PI_OAD(), PI_OSM(), PI_OMA() and PI_OMR()) can be called immediately.

After open-loop analog motion was done with PI_OAD(), the PI_RNP() function must be called before the servo can be switched on with PI_SVO() for closed-loop operation.

In open-loop operation, PI_RNP() must be called each time the motion mode is to be changed from nanostepping (PI_OSM(), PI_OMA(), PI_OMR()) to analog (PI_OAD()) motion and vice versa.

The following actions can take up to four times the slewrate value (parameter ID 0x7000002):

- Switching the servo on or off
- The first call of PI_OAD() after a change of motion mode
- The first call of PI_OSM() or PI_OMA() or PI_OMR() after a change of motion mode
- The PI_RNP() procedure

If a required transition between different motion modes was omitted, the E-861 sends an error message.

You can query the current state of the system (E-861 and NEXACT® linear drive) using the PI_qSRG(). With PI_IsControllerReady(), you can query if the controller is ready to perform a new command. If the controller responds with °, it is not ready.

While a joystick connected to the E-861 is enabled with PI_JON(), the servo can be switched on or off for the axis, and no transition is required. The axis can be controlled immediately by the joystick.

2.7. Samples

There are various sample programs for different programming languages to be found in the \Sample directory of the E-861 CD. The sample code below can also be found on the CD.

2.7.1. Stand-Alone E-861 System

The following example shows how to connect to an E-861, and (without the call printf()) represents a typical initialization, first for open-loop and then for closed-loop systems.

```
char axis[17];
BOOL blsMoving = TRUE;
BOOL bRtoSuccess;
BOOL bFlag;
int iVal,iChnl;
double dPos;
int ID;
// connect to the E-861 over RS-232 (COM port 1, baudrate 38400)
ID = PI_ConnectRS232 (1, 38400);
if (ID<0)
    return FALSE;
// Get the name of the connected axis
if (!PI_qSAL(ID, axis, 16))
    return FALSE;
// E-861 open-loop
// Switch off the Servo
bFlag = FALSE;
if(!PI_SVO(ID, axis,&bFlag))
    return FALSE;
bFlag = FALSE;
while(bFlag != TRUE)
{
    if(!PI_IsControllerReady(ID, &bFlag))
        return FALSE;
}
iVal = 10;
iChnl = 1;
if(!PI_OSM(ID,&iChnl, &iVal,1))
    return FALSE;
// Wait until the open-loop move is done.
blsMoving = TRUE;
while(blsMoving == TRUE)
{
    if(!PI_qPOS(ID, axis, &dPos))
        return FALSE;
    if(!PI_IsMoving(ID, axis, &blsMoving))
        return FALSE;
    printf("Pos: %g\n",dPos);
}
dPos = 0.0;
if(!PI_RNP(ID,&iChnl,&dPos, 1))
    return FALSE;
bFlag = FALSE;
while(bFlag != TRUE)
{
    if(!PI_IsControllerReady(ID, &bFlag))
        return FALSE;
}
for(double dVoltage = -50.0;dVoltage < 51.0;dVoltage+=10.0)
{
    if(!PI_OAD(ID, &iChnl, &dVoltage, 1))
        return FALSE;
    for (iVal = 0;iVal < 10; iVal++)
```

```

{
    if(!PI_qPOS(ID, axis, &dPos))
        return FALSE;
    printf("%fV -> Pos: %g\n", dVoltage, dPos);
    Sleep(10);
}
}
if(!PI_RNP(ID, &iChnl, &dPos, 1))
    return FALSE;
bFlag = FALSE;
while(bFlag != TRUE)
{
    if(!PI_IsControllerReady(ID, &bFlag))
        return FALSE;
}
printf("Open loop finished. Press a key\n");
getch();
// E-861 closed-loop system only: calling PI_MOV() to move the axis absolute
// will fail if an axis has not been referenced.
// Switch on the Servo
bFlag = TRUE;
if(!PI_SVO(ID, axis, &bFlag))
    return FALSE;
bFlag = FALSE;
while(bFlag != TRUE)
{
    if(!PI_IsControllerReady(ID, &bFlag))
        return FALSE;
}
// Reference the axis either using the refence switch, the negative or the positive limit switch -
// in this example the reference switch is used.
if(!PI_FRF(ID, axis))
    return FALSE;
// Wait until the reference move is done.
bFlag = FALSE;
while(bFlag != TRUE)
{
    if(!PI_IsControllerReady(ID, &bFlag))
        return FALSE;
}
// Now you can move the axis.
dPos = 2.0;
if(!PI_MOV(ID, axis, &dPos))
    return FALSE;
// Wait until the closed loop move is done.
bIsMoving = TRUE;
while(bIsMoving == TRUE)
{
    if(!PI_qPOS(ID, axis, &dPos))
        return FALSE;
    if(!PI_IsMoving(ID, axis, &bIsMoving))
        return FALSE;
    printf("Pos: %g\n", dPos);
}
PI_CloseConnection(ID);

```

2.7.2. E-861 System(s) in a Daisy Chain

The following example shows how to connect to an E-861 which is part of a daisy chain.

```

int IDs[16];
int DaisyChainID;
int iNumberOfConnectedDevices;
int iCounter;
int iActuallyConnected = 0;
char buffer[100];
// connect to the E-861 over RS-232 daisy chain (COM port 1, baudrate 38400)
DaisyChainID = PI_OpenRS232DaisyChain(1, 38400, &iNumberOfConnectedDevices, NULL, 0);

if (DaisyChainID < 0) // maybe the wrong baudrate or COM port was used?
    return FALSE;

// if there is no E-861 connected to the daisy chain, close it again and return.
if(iNumberOfConnectedDevices <= 0)

```

```

{
    PI_CloseDaisyChain(DaisyChainID);
    return FALSE;
}

// if there is at least one E-861 connected to the daisy chain (iNumberOfConnectedDevices > 0)
// try all possible addresses
for(iCounter = 1; iCounter <=16; iCounter++)
{
    IDs[iActuallyConnected] = PI_ConnectDaisyChainDevice(DaisyChainID, iCounter);
    if (IDs[iActuallyConnected] >=0)
    {
        if(!PI_qIDN(IDs[iActuallyConnected],buffer,99))
            return FALSE;
        printf("Connected to %s on Daisy Chain Address %d\n",buffer, iCounter);
        iActuallyConnected++;
    }
    else
        printf("No d.c. device on address %d\n",iCounter);
}

// now you can access the controllers
// ...
for(iCounter = 1; iCounter <= iActuallyConnected; iCounter++)
{
    PI_CloseConnection(IDs[iCounter]);
}
PI_CloseDaisyChain(DaisyChainID);

```

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

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");
```

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");
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. 12)). 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. 71).

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

NOTE

To connect the E-861 to the host PC, you can use the RS-232 **or** the USB 1.0 interface of the E-861. On the controller only one interface can be active at a time. The controller address and the baud rate of the E-861 are set via the DIP switches on the front panel (see the E-861 User manual for more information). All controllers in a daisy chain must be set to the same baudrate.

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

- `PI_InterfaceSetupDlg()`
- `PI_ConnectRS232()`
- `PI_ConnectDaisyChainDevice()`
- `PI_ConnectUSB()`
- `PI_ConnectUSBWithBaudRate()`.

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.

NOTE

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.

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.

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

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.25) will close the connection to the specified controller and free its system resources.

The communications functions are listed below:

Function	Short Description	Page
void PI_CloseConnection (int ID)	Close connection to the controller	14
void PI_CloseDaisyChain (int iPortId)	Close connection to the daisy chain port	14
int PI_ConnectDaisyChainDevice (int iPortId, int iDeviceNumber)	Open a daisy chain device	15
int PI_ConnectRS232 (int iPortNumber, int iBaudRate)	Open an RS-232 ("COM") interface to a controller	15
int PI_ConnectUSB (const char* szDescription)	Open an USB connection to a	15

Function	Short Description	Page
	controller using one of the identification strings listed by PI_EnumerateUSB()	
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()	15
int PI_EnumerateUSB (char* szBuffer, int iBufferSize, const char* szFilter)	Lists the identification strings of all controllers available via USB interfaces	15
int PI_GetError (int ID)	Get error status of the DLL and, if clear, that of the controller	16
int PI_InterfaceSetupDlg (const char* szRegKeyName)	Open dialog to let user select the interface and create a new PI object	16
BOOL PI_IsConnected (int ID)	Check if there is a controller with an ID of <i>ID</i>	16
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	16
long PI_OpenUSBdaisyChain (const char* szDescription, long* pNumberOfConnectedDaisyChainDevices, char* szDeviceIDNs, long iBufferSize);	Open a USB interface to a daisy chain	17
BOOL PI_SetErrorCheck (int ID, BOOL bErrorCheck)	Set error-check mode of the library	17
BOOL PI_TranslateError (int iErrorNumber, char* szErrorMessage, int iBufferSize)	Translate error number to error message	17

4.2. Function Description

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_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. 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_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_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.

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 9600 baud 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.

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_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_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.71) 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. 71) 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. 25) 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.

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, long 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 *pNumberOfConnectedDaisyChainDevices*.

Arguments:

szDescription the description of the controller returned by `PI_EnumerateUSB`

pNumberOfConnectedDaisyChainDevices 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

4.3. Interface Settings

With `PI_InterfaceSetupDlg()`, p. 16, 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.

5. Functions for Sending and Reading Strings

With PI library functions for GCS query commands the controller automatically continues processing following 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 Section "Basic Functions for GCS Commands".

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. 11)

```
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. 11)

```
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. 11)

6. Basic Functions for GCS Commands

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.
See “Functions for Sending and Reading Strings” (p. 18) for details.

6.1. Overview

For the GCS commands belonging to the appropriate functions see Section 6.2 on p. 25 and the E-861 User manual.

Function	Short Description	Page
BOOL PI_ACC (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>)	Set Closed-Loop Acceleration	25
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	25
BOOL PI_DEC (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>)	Set Closed-Loop Deceleration	25
BOOL PI_DEL (int <i>ID</i> , double <i>dmSeconds</i>)	Delay The Command Interpreter	26
BOOL PI_DIO (long <i>ID</i> , const int* <i>piChannelsArray</i> , const BOOL* <i>pbValueArray</i> , int <i>iArraySize</i>)	Set Digital Output Lines	26
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	26
BOOL PI_DRT (int <i>ID</i> , const int* <i>piRecordTableIdsArray</i> , const char* <i>szTriggerSourceArray</i> , const double* <i>pdValueArray</i>)	Set Data Recorder Trigger Source	27
BOOL PI_FED (int <i>ID</i> , const char* <i>szAxes</i> , const int* <i>piEdgeArray</i> , const int* <i>piParamArray</i>)	Find Edge	27
BOOL PI_FNL (int <i>ID</i> , const char* <i>szAxes</i>)	Fast Move To Negative Limit	28
BOOL PI_FPL (int <i>ID</i> , const char* <i>szAxes</i>)	Fast Move To Positive Limit	28
BOOL PI_FRF (int <i>ID</i> , const char* <i>szAxes</i>)	Fast Move To Reference Switch	29
BOOL PI_GetAsyncBuffer (int <i>ID</i> , double ** <i>pnValArray</i>)		29
int PI_GetAsyncBufferIndex (int <i>ID</i>)		29
BOOL PI_GOH (int <i>ID</i> , const char* <i>szAxes</i>)	Go To Home Position	29
BOOL PI_HLT (int <i>ID</i> , const char* <i>szAxes</i>)	Halt Motion Smoothly	30
BOOL PI_IsControllerReady (const int <i>ID</i> , BOOL* <i>pbValueArray</i>)	Asks controller for ready status	30
BOOL PI_IsMoving (int <i>ID</i> , const char* <i>szAxes</i> , BOOL* <i>pbValueArray</i>)	Check if axes are moving	30
BOOL PI_IsRunningMacro (int <i>ID</i> , BOOL* <i>pbRunningMacro</i>)	Check if controller is currently running a macro	30
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	31
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	31
BOOL PI_JON (int <i>ID</i> , const int* <i>iJoystickIDsArray</i> , const	Set Joystick Activation Status	31

Function	Short Description	Page
BOOL* pbValArray, int iArraySize)		
BOOL PI_MAC_BEG (int ID, const char * szName)	Call Macro Function: Start recording macro	31
BOOL PI_MAC_DEF (int ID, const char * szMacroName)	Call Macro Function: Set the specified macros as start-up macro	32
BOOL PI_MAC_DEL (int ID, const char * szMacroName)	Call Macro Function: Delete macro	32
BOOL PI_MAC_END (int ID)	Call Macro Function: End macro recording	32
BOOL PI_MAC_NSTART (int ID, const char * szName, int nrRuns)	Call Macro Function: Execute macro n times	33
BOOL PI_MAC_qDEF (int ID, char * szBuffer, int iBufferSize)	Call Macro Function: Ask name of start-up macro	33
BOOL PI_MAC_START (int ID, const char * szName)	Call Macro Function: Start macro (single run)	33
BOOL PI_MEX (int ID, const char * szCondition)	Stop Macro Execution Due To Condition	33
BOOL PI_MOV (int ID, const char* szAxes, const double* pdValueArray)	Set Target Position	34
BOOL PI_MVR (int ID, const char* szAxes, const double* pdValueArray)	Set Target Relative To Current Position	34
BOOL PI_OAC (long ID, const int* piPIEZOWALKChannelsArray, const double* pdValueArray, int iArraySize);	Set Open-Loop Acceleration	34
BOOL PI_OAD (int ID, const int* piPIEZOWALKChannelsArray, const double* pdValueArray, int iArraySize)	Open-Loop Analog Driving	35
BOOL PI_ODC (long ID, const int* piPIEZOWALKChannelsArray, const double* pdValueArray, int iArraySize);	Set Open-Loop Deceleration	35
BOOL PI_OMA (long ID, const char* szAxes, const double* pdValueArray);	Absolute Open-Loop Motion	36
BOOL PI_OMR (long ID, const char* szAxes, const double* pdValueArray);	Relative Open-Loop Motion	36
BOOL PI_OSM (int ID, const int* piPIEZOWALKChannelsArray, const int* piValueArray, int iArraySize)	Open-Loop Step Moving (using full step cycles)	37
BOOL PI_OSMf (long ID, const int* piPIEZOWALKChannelsArray, const double* pdValueArray, int iArraySize);	Open-Loop Step Moving (allowing also parts of a step cycle)	38
BOOL PI_OVL (int ID, const int* piPIEZOWALKChannelsArray, double* pdValueArray, int iArraySize)	Set Open-Loop Velocity	39
BOOL PI_POS (int ID, const char* szAxes, const double* pdValueArray)	Set Real Position	39
BOOL PI_qACC (int ID, const char* szAxes, double* pdValueArray)	Get Closed-Loop Acceleration	39
BOOL PI_qCST (int ID, const char* szAxes, char* szNames, int iBufferSize)	Get Stage Type Of Selected Axis	40
BOOL PI_qCSV (int ID, double* pdCommandSyntaxVersion)	Get Current Syntax Version	40
BOOL PI_qDEC (int ID, const char* szAxes, double* pdValueArray)	Get Closed-Loop Deceleration	40

Function	Short Description	Page
BOOL PI_qDIO (long <i>ID</i> , const long* <i>piChannelsArray</i> , BOOL* <i>pbValueArray</i> , int <i>iArraySize</i>)	Get Digital Input Lines	40
BOOL PI_qDRC (int <i>ID</i> , const int* <i>piRecordTableIdsArray</i> , char* <i>szRecordSource</i> , int* <i>iRecordOptionsArray</i> , int <i>iArraySize</i>)	Get Data Recorder Configuration	41
BOOL PI_qDRR (int <i>ID</i> , const int* <i>piRecTableIdArrays</i> , int <i>iNumberOfRecTables</i> , int <i>iOffset</i> , int <i>nrValues</i> , double** <i>pdValArray</i> , char* <i>szGcsArrayHeader</i> , int <i>iGcsArrayHeaderMaxSize</i>)	Get Recorded Data Values	41
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	42
BOOL PI_qDRT (int <i>ID</i> , const int* <i>piRecordTableIdsArray</i> , const char* <i>szTriggerSourceArray</i> , double* <i>pdValueArray</i> , int <i>iArraySize</i>)	Get Data Recorder Trigger Source	42
BOOL PI_qERR (int <i>ID</i> , int* <i>piError</i>)	Get Error Number	42
BOOL PI_qFRF (int <i>ID</i> , const char* <i>szAxes</i> , int* <i>piResult</i>)	Get Referencing Result	43
BOOL PI_qHDR (int <i>ID</i> , char* <i>szBuffer</i> , int <i>iBufferSize</i>)	Get All Data Recorder Options	43
BOOL PI_qHLP (int <i>ID</i> , char* <i>szBuffer</i> , int <i>iBufferSize</i>)	Get List of Available Commands	43
BOOL PI_qHPA (int <i>ID</i> , char* <i>szBuffer</i> , int <i>iBufferSize</i>)	Get List of Available Parameters	43
BOOL PI_qIDN (int <i>ID</i> , char* <i>szBuffer</i> , int <i>iBufferSize</i>)	Get Device Identification	44
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	44
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	44
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	45
BOOL PI_qJON (int <i>ID</i> , const int* <i>iJoystickIDsArray</i> , BOOL* <i>pbValarray</i> , int <i>iArraySize</i>)	Get Joystick Activation Status	45
BOOL PI_qLIM (int <i>ID</i> , const char* <i>szAxes</i> , BOOL* <i>pbValueArray</i>)	Indicate Limit Switches	45
BOOL PI_qMAC (int <i>ID</i> , char* <i>szName</i> , char* <i>szBuffer</i> , const int <i>maxlen</i>)	List Macros	45
BOOL PI_qMOV (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>)	Get Target Position	46
BOOL PI_qOAC (long <i>ID</i> , const int* <i>piPIEZOWALKChannelsArray</i> , double* <i>pdValueArray</i> , int <i>iArraySize</i>);	Get Open-Loop Acceleration	46
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	46
BOOL PI_qODC (long <i>ID</i> , const int* <i>piPIEZOWALKChannelsArray</i> , double* <i>pdValueArray</i> , int <i>iArraySize</i>);	Get Open-Loop Deceleration	46
BOOL PI_qOMA (long <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>);	Get Open-Loop Target Position	47
BOOL PI_qONT (int <i>ID</i> , const char* <i>szAxes</i> , BOOL* <i>pbValueArray</i>)	Get On Target State	47
BOOL PI_qOVL (int <i>ID</i> , const int* <i>piPIEZOWALKChannelsArray</i> , double* <i>pdValueArray</i>)	Get Open-Loop Velocity	47
BOOL PI_qPOS (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>)	Get Real Position	48
BOOL PI_qRMC (int <i>ID</i> , char* <i>szBuffer</i> , const int <i>iBufferSize</i>)	List Running Macros	48

Function	Short Description	Page
BOOL PI_qRON (const int ID, char *const szAxes, BOOL *pbValarray)	Get Reference Mode	48
BOOL PI_qRTR (int ID, int iRecordTableRate)	Get Record Table Rate	48
BOOL PI_qSAI (int ID, char* szAxes, int iBufferSize)	Get List Of Current Axis Identifiers	48
BOOL PI_qSAI_ALL (int ID, char* szAxes, int iBufferSize)	Get List Of Current Axis Identifiers	49
BOOL PI_qSEP (int ID, const char* szAxes, const int* piParameterArray, double* pdValueArray, char* szStrings, int iMaximumStringSize)	Get Nonvolatile Memory Parameters	49
BOOL PI_qSPA (int ID, const char* szAxes, const int* piParameterArray, double* pdValueArray, char* szStrings, int iMaximumStringSize)	Get Volatile Memory Parameters	49
BOOL PI_qSRG (int ID, const char*szAxes, const int* iRegisterArray, int * iValArray)	Query Status Register Value	50
BOOL PI_qSSA (int ID, const int* iPIEZOWALKChannels, double* pdValueArray, int iArraySize)	Gets current value of voltage amplitude used for nanostepping motion	50
BOOL PI_qSVO (int ID, const char* szAxes, BOOL* pbValueArray)	Get Servo State (Open-Loop / Closed-Loop Operation)	51
BOOL PI_qTAC (int ID, int * pnNr)	Tell Analog Channels	51
BOOL PI_qTAV (int ID, const char* szAnalogInputChannels, int* piValueArray)	Get Analog Input Voltage	51
BOOL PI_qTIO (int ID, int* piInputNr, int* piOutputNr)	Tell Digital I/O Lines	51
BOOL PI_qTMN (int ID, const char* szAxes, double* pdValueArray)	Get Minimum Commandable Position	52
BOOL PI_qTMX (int ID, const char* szAxes, double* pdValueArray)	Get Maximum Commandable Position	52
BOOL PI_qTNR (int ID, int* piNumberOfRecordTables)	Get Number Of Record Tables	52
BOOL PI_qTRS (int ID, const char* szAxes, BOOL * pbValueArray)	Indicate Reference Switch	52
BOOL PI_qVEL (int ID, const char* szAxes, double* pdValueArray)	Get Closed-Loop Velocity	52
BOOL PI_RBT (int ID)	Reboot System	53
BOOL PI_RNP (int ID, const int* piPIEZOWALKChannelsArray, const double* pdValueArray, int iArraySize)	Relax PiezoWalk Piezos	53
BOOL PI_RON (const int ID, char *const szAxes, BOOL *pbValarray)	Set Reference Mode	53
BOOL PI_RPA (int ID, const char* szAxes, const int* piParameterArray)	Reset Volatile Memory Parameters	54
BOOL PI_RTR (int ID, int iRecordTableRate)	Set Record Table Rate	54
BOOL PI_SEP (int ID, const char* szPassword, const char* szAxes, const int* piParameterArray, const double* pdValueArray, const char* szStrings)	Set Nonvolatile Memory Parameters	54
BOOL PI_SPA (int ID, const char* szAxes, const int* piParameterArray, double* pdValueArray, const char* szStrings)	Set Volatile Memory Parameters	55
BOOL PI_SSA (int ID, const int* piPIEZOWALKChannelsArray, const double* pdValueArray, int iArraySize)	Set Step Amplitude	56
BOOL PI_STE (int ID, const char* szAxes, double* pdStepSize)	Start Step And Response – Measurement	56

Function	Short Description	Page
BOOL PI_STP (int <i>ID</i> , const char* <i>szAxes</i>)	Stop All Motion	56
BOOL PI_SVO (int <i>ID</i> , const char* <i>szAxes</i> , BOOL* <i>pbValueArray</i>)	Set Servo State (Open-Loop / Closed-Loop Operation)	57
BOOL PI_VEL (int <i>ID</i> , const char* <i>szAxes</i> , double* <i>pdValueArray</i>)	Set Closed-Loop Velocity	57
BOOL PI_WAC (int <i>ID</i> , char * <i>szCondition</i>)	Wait For Condition For Macro Execution	57
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	58

6.2. Function Description

See “Function Calls” (p. 11) for some general notes about the parameter syntax.

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).

The lowest possible value for *pdValueArray* is 0.

The maximum value which can be set with the PI_ACC() is given by the Maximum closed-loop acceleration parameter, ID 0x4A

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray maximum accelerations for the axes

Returns:

TRUE if no error, FALSE otherwise (see p. 11)

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 "" on p. 59) and sends them to the controller so that the controller parameters (see p. 64) are properly adjusted to the connected mechanics. 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

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. 11)

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).

The lowest possible value for *pdValueArray* is 0.

The maximum value which can be set with the PI_DEC() is given by the Maximum closed-loop deceleration parameter, ID 0x4B.

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray maximum decelerations for the axes

Returns:

TRUE if no error, FALSE otherwise (see p. 11)

BOOL PI_DEL (int *ID*, double *dmSeconds*)

Corresponding command: DEL

Delay the controller for *dmSeconds* milliseconds.

PI_DEL() can only be used in macros. Do not mistake PI_MAC_DEL() which deletes macros for PI_DEL() which delays.

See the PI_MAC... () functions and "Working with Controller Macros" in the controller User manual for more information.

Arguments:

ID ID of controller

dmSeconds delay value in milliseconds

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding GCS command: DIO

Set digital output channels HIGH or LOW. If *pbValueArray*[index] is **TRUE** the mode is set to **HIGH**, otherwise it is set to **LOW**.

Note: With the E-861, you can either set a single line per DIO command, or all lines at once. The identifiers to use for the lines are 1 to 4. With the identifier 0, all lines are set according to a bit pattern given (in decimal or hexadecimal format) by *pbValueArray*.

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 *piChannelsArray* and *pbValueArray*

Returns:

TRUE if successful, **FALSE** otherwise

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.

The E-861 has two data recorder tables with 1024 points per 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 meaning of this parameter depends on the corresponding record option.

piRecordOptionsArray can be one of:

0 = nothing is recorded

1 = target position (record source = axis)

2 = real position (record source = axis)

3 = position error (record source = axis)

70 = Commanded Velocity (open-loop or closed-loop velocity, depending on the current servo state) (record source = axis)

71 = Commanded Acceleration (open-loop or closed-loop acceleration, depending on the current servo state) (record source = axis)

73 = Motor Output (record source = axis)

80 = Signal Status Register (record source = axis)

81 = Analog Input (record source = input channel)

Returns:

TRUE if no error, FALSE otherwise (see p. 11)

BOOL **PI_DRT** (int *ID*, const int* *piRecordTableIdsArray*, const char* *szTriggerSourceArray*, const double* *pdValueArray*)

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 (axes or input channels) and record options to the recorder tables see PI_DRC().

For more information regarding data recording see the notes in PI_qDRR.

Arguments:

ID ID of controller

piRecordTableIdsArray ID of the record table

szTriggerSourceArray ID of the trigger source, can be

0 = default setting; data recording is triggered with any command which changes position (move command);

1 = any command changing position (e.g. move functions like PI_MVR(), PI_MOV(), PI_STE(), PI_OAD(), PI_OSM(), PI_OMA(), PI_OMR())

2 = next command; only valid until the very next command was sent, afterwards the trigger source ID is reset to the default value (0)

6 = any command changing target position (e.g. MVR, MOV), resets trigger after execution

pdValueArray value depends on the trigger source ID; for *szTriggerSourceArray* 0, 1, 2, 6 and 7

pdValueArray must contain a dummy, e.g. an arbitrary character

Returns:

TRUE if no error, FALSE otherwise (see p. 11)

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.

This function can be used both with servo enabled or disabled for the commanded axis.

Caution: Do not use this command if no sensor is present in your system. Otherwise the connected mechanics can run into the hardstop at full speed which may cause damage to your hardware setup.

In contrast to PI_FNL(), PI_FPL() and PI_FRF(), this command does not change the reference state of the axis and does not set a certain position value at the selected edge.

If multiple axes are given in the command, they are moved synchronously.

The controller firmware detects the presence or absence of reference switch and limit switches using controller parameters (ID 0x14 for reference switch; ID 0x32 for limit switches). According to the values of those parameters, the E-861 enables or disables PI_FED() motions to the appropriate signal edges.

Adapt the parameter values to your hardware using PI_SPA() or PI_SEP(). See "System Parameter Overview" on p. 64 for more information.

PI_FED() can be used in combination with PI_qPOS() to measure the physical travel range of a new mechanics and thus to identify the values for the corresponding controller parameters: the distance from negative to positive limit switch, the distance between the negative limit switch and the reference switch (DISTANCE_REF_TO_N_LIM, parameter ID 0x17), and the distance between reference switch and positive limit switch (DISTANCE_REF_TO_P_LIM, parameter ID 0x2F). See "Travel Range Adjustment" in the controller User manual for more information.

The motion can be interrupted by PI_STP() and PI_HLT().

Motion commands like PI_FED() are not allowed when the joystick is active for the axis. See "Joystick Control" in the controller User manual for details.

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 not needed with E-861, should contain zeros

Returns:

TRUE if successful, **FALSE** otherwise

Errors:

PI_UNKNOWN_AXIS_IDENTIFIER cAxis is not a valid axis identifier

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

Corresponding command: FNL

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.

This function can be used both with servo enabled or disabled for the commanded axis.

Caution: Do not use this command if no sensor is present in your system. Otherwise the connected mechanics can run into the hardstop at full speed which may cause damage to your hardware setup.

The implementation of the stop-behaviour (e.g. limit switch stop or hardstop) and the position which is set (e.g. 0 or min. range limit) depend on the controller.

PI_FNL() can be used for referencing of multiple axes which have no reference switch. 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 get the result from the controller. Use PI_STP() to stop referencing.

Notes:

Calling the PI_FNL() function resets the current positions of the axes. Therefore moving the axes to the same position using PI_MOV() (absolute move) before and after a call of this function may move the mechanics to a different physical position! You should call PI_FNL() only once after controller power-on or reboot.

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_FPL (int *ID*, const char* *szAxes*)

Corresponding command: FPL

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.

This function can be used both with servo enabled or disabled for the commanded axis.

Caution: Do not use this command if no sensor is present in your system. Otherwise the connected mechanics can run into the hardstop at full speed which may cause damage to your hardware setup.

The implementation of the stop-behaviour (e.g. limit switch stop or hardstop) and the position which is set (range limit) depend on the controller.

PI_FPL() can be used for referencing of multiple axes which have no reference switch. 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 get the result from the controller. Use PI_STP() to stop referencing.

Notes:

Calling the PI_FPL() function resets the current positions of the axes. Therefore moving the axes to the same position using PI_MOV() (absolute move) before and after a call of this function may move the mechanics to a different physical position! You should call PI_FPL() only once after controller power-on or reboot.

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_FRF (int *ID*, const char* *szAxes*)

Corresponding command: FRF

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.

This function can be used both with servo enabled or disabled for the commanded axis.

Caution: Do not use this command if no sensor is present in your system. Otherwise the connected mechanics can run into the hardstop at full speed which may cause damage to your hardware setup.

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 get the result from the controller. Use `PI_STP()` to stop referencing.

Notes:

Calling the `PI_FRF()` function resets the current positions of the axes. Therefore moving the axes to the same position using `PI_MOV()` (absolute move) before and after a call of this function may move the mechanics to a different physical position! You should call `PI_FRF()` only once after controller power-on or reboot.

Arguments:

ID ID of controller
szAxes string with axes

Returns:

TRUE if successful, **FALSE** otherwise

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

Get address of internal buffer used for storing data read in by a call to `PI_qDRR()`.

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()`.

Arguments:

ID ID of controller

Returns:

Index of the data element which was last read in, -1 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()`).

Caution: Do not use this command if no sensor is present in your system. Otherwise the connected mechanics can run into the hardstop at full speed which may cause damage to your hardware setup.

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. 11)

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

Corresponding **command**: HLT

Halt the motion of given axes smoothly. Error code 10 is set. After the stage was stopped, if servo is on the target position is set to the current position.

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. 11)

BOOL PI_IsControllerReady (const int *ID*, BOOL * *pbValue*)

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

pbValue status of the controller:

B1h (ASCII character 177 = "±" in Windows) if controller is ready

B0h (ASCII character 176 = "°" in Windows) if controller is not ready (e.g. performing a referencing command)

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_IsMoving (const int *ID*, char *const *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.

Only effective in closed-loop operation (servo ON).

Arguments:

ID ID of controller

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

pbValueArray 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 ID of the joystick axes (must be 1 for E-861, which supports only 1 joystick axis per controller)

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_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_MAC_BEG (int ID, const char * szName)

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 "Working with Controller Macros" and the MAC command description in the controller User manual for details.

Arguments:

ID ID of controller

szName name under which macro will be stored in the controller

Returns:

TRUE if successful, **FALSE** otherwise

Errors:

PI_IN_MACRO_MODE if a macro is already being recorded

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

Corresponding command: MAC DEF

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

See "Working with 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 DEL

Delete macro with name *szName*. To find out what macros are available call PI_qMAC().

See "Working with 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 "Working with 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

Errors:

PI_NOT_IN_MACRO_MODE the controller was not recording a macro

BOOL PI_MAC_NSTART (int ID, const char * *szName*, int *nrRuns*)**Corresponding command:** MAC START

Start macro with name *szName*. The macro is repeated *nrRuns* times. To find out what macros are available call PI_qMAC().

See "Working with Controller Macros" and the MAC command description in the controller User manual for details.

Arguments:

ID ID of controller

szName string with name of the macro to start

nrRuns nr of runs

Returns:

TRUE if successful, **FALSE** otherwise

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

Ask for the start-up macro.

See "Working with 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 *buffer*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_MAC_START (int ID, const char * *szMacroName*)**Corresponding command:** MAC START

Start macro with name *szName*. To find out what macros are available call PI_qMAC().

See "Working with 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_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 "Working with Controller Macros" and the MEX command description in the controller User manual for details and examples.

Arguments:

ID ID of controller

szCondition string with condition to evaluate

Returns:

TRUE if successful, **FALSE** otherwise

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.

Caution: Do not use this command if no sensor is present in your system. Otherwise the connected mechanics can run into the hardstop at full speed which may cause damage to your hardware setup.

Motion commands like PI_MOV are not allowed when a joystick is active on the axis.

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray target positions for the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 11)

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.

Caution: Do not use this command if no sensor is present in your system. Otherwise the connected mechanics can run into the hardstop at full speed which may cause damage to your hardware setup.

Motion commands like PI_MOV are not allowed when a joystick is active on the axis.

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. 11)

BOOL PI_OAC (long *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).

Acceleration can be changed while the axis is moving.

The lowest possible value for *pdValueArray* is 0.

PI_OAC changes the value of the Current open-loop acceleration parameter (ID 0x7000202) in volatile memory (can be saved as power-on default with PI_WPA, can also be changed with PI_SPA and PI_SEP).

The maximum value which can be set with PI_OAC is given by the Maximum open-loop acceleration parameter, ID 0x7000205 (can be changed with PI_SPA and PI_SEP)

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).

Analog driving means a motion with highest motion resolution. All piezos are in contact with the runner and move in the same direction. To achieve this, all the piezos are driven by the same "feed" voltage. The allowable range is between -55 and 55. The sign determines the motion direction.

In open-loop operation, PI_RNP() must be called each time the motion mode is to be changed from nanostepping motion (PI_OSM(), PI_OMA(), PI_OMR()) to analog motion (PI_OAD()) and vice versa. PI_RNP() brings the drive to a full-holding-force, zero-drive-voltage Relaxed state.

The first PI_OAD() called for a NEXACT® linear drive which is in the Relaxed state prepares the drive for analog motion (brings it to the Analog state) before the actual motion is done. Once the drive is in the Analog state, each subsequent PI_OAD() motion will be executed immediately.

The first PI_OAD() after a change of motion mode and the PI_RNP() procedure can take up to four times the slewrate value (parameter ID 0x7000002; can be changed with PI_SPA() and PI_SEP()).

After open-loop analog motion was done with PI_OAD(), PI_RNP() must be called before the servo can be switched on with PI_SVO() for closed-loop operation.

You can query the current state of the system (E-861 and NEXACT® linear drive) using PI_qSRG().

See "Modes of Operation" in the E-861 User manual for more information and Section 2.4 on p. 6 for an example.

Motion commands like PI_OAD() are not allowed when a joystick is active on the axis. See "Joystick Control" in the E-861 User manual for details.

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. 11)

BOOL PI_ODC (long *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).

Deceleration can be changed while the axis is moving.

The lowest possible value for *pdValueArray* is 0.

PI_ODC changes the value of the Current open-loop deceleration parameter (ID 0x7000203) in volatile memory (can be saved as power-on default with PI_WPA, can also be changed with PI_SPA and PI_SEP).

The maximum value which can be set with PI_ODC is given by the Maximum open-loop deceleration parameter, ID 0x7000206 (can be changed with PI_SPA and PI_SEP)

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 (long *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).

Caution: Do not use this command if no sensor is present in your system. Otherwise the connected mechanics can run into the hardstop at full speed which may cause damage to your hardware setup.

With PI_OMA there is no position control (i.e. the target position is not maintained by any control loop). Due to the motion profile (velocity, acceleration), there will typically be an overshoot. The E-861 counts the overshoot steps, and the axis moves back the corresponding number of steps.

The velocity for open-loop nanostepping motion depends on the step size and on the step frequency. The step size is given by the amplitude of the feed voltage which can be set with PI_SSA and queried with PI_qSSA. The step frequency can be set with PI_OVL and queried with PI_qOVL.

In open-loop operation, an PI_RNP function must be sent each time the motion mode is to be changed from nanostepping motion (PI_OMA, PI_OMR or PI_OSMf) to analog motion (PI_OAD) and vice versa. RNP brings the drive to a full-holding-force, zero-drive-voltage Relaxed state. See "Modes of Operation" and "Example for Commanding Motion" in the E-861 User manual for more information and an example.

The first PI_OMA sent for a NEXACT® linear drive which is in the Relaxed state prepares the drive for nanostepping motion (brings it to the In Motion state) before the actual motion is done. Once the drive is in the In Motion state, each subsequent PI_OMA motion will be executed immediately.

Note that a second PI_OMA function cannot be processed as long as a motion commanded with a first PI_OMA or PI_OMR is still performed.

The first PI_OMA after a change of motion mode and the RNP procedure can take up to four times the slewrate value (parameter ID 0x7000002; can be changed with PI_SPA and PI_SEP).

You can query the current state of the system (E-861 and NEXACT® linear drive) using PI_qSRG. See "Changing Motion and Servo Mode" in the E-861 User manual for more information.

The motion can be interrupted by PI_STP and PI_HLT.

Motion functions like PI_OMA are not allowed when a joystick is active on the axis. See "Joystick Control" in the E-861 User manual for details.

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray target positions for the axes

Returns:

TRUE if no error, FALSE otherwise (see p. 11)

BOOL PI_OMR (long *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).

Caution: Do not use this command if no sensor is present in your system. Otherwise the connected mechanics can run into the hardstop at full speed which may cause damage to your hardware setup.

With PI_OMR there is no position control (i.e. the target position is not maintained by a control loop). Due to the motion profile (velocity, acceleration), there will typically be an overshoot. The E-861 counts the

overshoot steps, and the axis moves back the corresponding number of steps.

The velocity for open-loop nanostepping motion depends on the step size and on the step frequency. The step size is given by the amplitude of the f voltage which can be set with PI_SSA and queried with PI_qSSA. The step frequency can be set with PI_OVL and queried with PI_qOVL.

In open-loop operation, an PI_RNP function must be sent each time the motion mode is to be changed from nanostepping motion (PI_OMA, PI_OMR or PI_OSMf) to analog motion (PI_OAD) and vice versa. RNP brings the drive to a full-holding-force, zero-drive-voltage Relaxed state. See "Modes of Operation" and "Example for Commanding Motion" in the E-861 User manual for more information and an example.

The first PI_OMR sent for a NEXACT® linear drive which is in the Relaxed state prepares the drive for nanostepping motion (brings it to the In Motion state) before the actual motion is done. Once the drive is in the In Motion state, each subsequent OMR motion will be executed immediately.

Note that a second PI_OMR function cannot be processed as long as a motion commanded with a first PI_OMR or PI_OMA is still performed.

The first PI_OMR after a change of motion mode and the RNP procedure can take up to four times the slewrate value (parameter ID 0x7000002; can be changed with PI_SPA and PI_SEP).

You can query the current state of the system (E-861 and NEXACT® linear drive) using PI_qSRG.

See "Changing Motion and Servo Mode" in the E-861 User manual for more information.

The motion can be interrupted by PI_STP and PI_HLT.

Motion functions like PI_OMR are not allowed when a joystick is active on the axis. See "Joystick Control" in the E-861 User manual for details.

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray target positions for the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 11)

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.

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

The difference between PI_OSM and the next function PI_OSMf is that with the latter parts of a step cycle can be commanded. In opposite to that, PI_OSM uses integer values for the number of steps and thus allows only complete steps for the PiezoWalk channels.

The NEXACT® linear drive performs the given number of steps of a pre-defined size and can thus cover the whole runner (i.e. the travel range is only limited by the length of the runner). This operating mode is also referred to as "nanostepping mode".

The velocity for open-loop step motion depends on the step size and on the step frequency. The step size is given by the amplitude of the feed voltage which can be set with PI_SSA() and queried with PI_qSSA(). The step frequency can be set with PI_OVL() and queried with PI_qOVL().

With each new call of PI_OSM(), the number of steps to perform is counted starting from zero (there is no adding up of steps).

In open-loop operation, PI_RNP() must be called each time the motion mode is to be changed from nanostepping motion (PI_OSM()) to analog motion (PI_OAD()) and vice versa. PI_RNP() brings the drive to a full-holding-force, zero-drive-voltage Relaxed state.

The first PI_OSM() called for a NEXACT® linear drive which is in the Relaxed state prepares the drive for nanostepping motion (brings it to the In Motion state) before the actual motion is done. Once the drive is in the In Motion state, each subsequent PI_OSM() motion will be executed immediately.

The first PI_OSM() after a change of motion mode and the PI_RNP() procedure can take up to four times the slewrate value (parameter ID 0x7000002; can be changed with PI_SPA() and PI_SEP()).

You can query the current state of the system (E-861 and NEXACT® linear drive) using PI_qSRG().

See "Modes of Operation" in the E-861 User manual for more information and Section 2.4 on p. 6 for an example.

The motion can be interrupted by PI_STP() and PI_HLT().

Motion commands like PI_OAD() are not allowed when a joystick is active on the axis. See "Joystick Control" in the E-861 User manual for details.

Arguments:

ID ID of controller

piPIEZOWALKChannelsArray array with PiezoWalk channels

pdValueArray 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. 11)

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.

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

The difference between PI_OSMf and the previous function PI_OSM is that with PI_OSMf parts of a step cycle can be commanded. In opposite to that, PI_OSM uses integer values for the number of steps and thus allows only complete steps for the PiezoWalk channels.

The NEXACT® linear drive performs the given number of steps of a pre-defined size and can thus cover the whole runner (i.e. the travel range is only limited by the length of the runner). This operating mode is also referred to as "nanostepping mode".

The velocity for open-loop step motion depends on the step size and on the step frequency. The step size is given by the amplitude of the feed voltage which can be set with PI_SSA() and queried with PI_qSSA(). The step frequency can be set with PI_OVL() and queried with PI_qOVL().

With each new call of PI_OSM(), the number of steps to perform is counted starting from zero (there is no adding up of steps).

In open-loop operation, PI_RNP() must be called each time the motion mode is to be changed from nanostepping motion (PI_OSM()) to analog motion (PI_OAD()) and vice versa. PI_RNP() brings the drive to a full-holding-force, zero-drive-voltage Relaxed state.

The first PI_OSM() called for a NEXACT® linear drive which is in the Relaxed state prepares the drive for nanostepping motion (brings it to the In Motion state) before the actual motion is done. Once the drive is in the In Motion state, each subsequent PI_OSM() motion will be executed immediately.

The first PI_OSM() after a change of motion mode and the PI_RNP() procedure can take up to four times the slewrate value (parameter ID 0x7000002; can be changed with PI_SPA() and PI_SEP()).

You can query the current state of the system (E-861 and NEXACT® linear drive) using PI_qSRG().

See "Modes of Operation" in the E-861 User manual for more information and Section 2.4 on p. 6 for an example.

The motion can be interrupted by PI_STP() and PI_HLT().

Motion commands like PI_OAD() are not allowed when a joystick is active on the axis. See "Joystick Control" in the E-861 User manual for details.

Arguments:

ID ID of controller

piPIEZOWALKChannelsArray array with PiezoWalk channels

pdValueArray number of steps for the PiezoWalk channels

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

Returns:

TRUE if no error, FALSE otherwise (see p. 11)

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).

PI_OVL() can be changed while the PiezoWalk channel is moving.

The velocity is positive or zero (max. 1.5 kHz for N-310.11 NEXACT® linear drive).

The velocity for open-loop nanostepping motion is also influenced by the step amplitude set with PI_SSA().

The maximum value which can be set with PI_OVL() is given by the Open-loop velocity parameter, ID 0x7000201 (can be changed with PI_SPA() and PI_SEP()).

On power-on, the current open-loop velocity is half the maximum.

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. 11)

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

Corresponding command: POS

Set current position for given axis. An axis is considered as "referenced" when the position was set with PI_POS, so that PI_qFRF replies "1".

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?

Get the current value of the closed-loop acceleration for *szAxes*.

For the current open-loop acceleration, query with PI_qOAC().

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. 11)

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.

The stage name is read from the Stage Name parameter (ID 0x3C) whose factory default value is "DEFAULT_STAGE-N".

You change the parameter value using PI_SPA or PI_SEP.

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. 11)

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 (1.0 for GCS 1.0, 2.0 for GCS 2.0).

Returns:

TRUE if no error, **FALSE** otherwise (see p. 11)

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

Corresponding command: DEC?

Get the current value of the closed-loop deceleration for *szAxes*.

E-861: For the current open-loop deceleration, query with PI_qODC().

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. 11)

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

Corresponding command: DIO?

Returns the states of the specified Digital Input channels.

Notes:

Use **PI_qTIO** (p. 51) to get the number of installed digital I/O channels.

You can either read a single line per PI_qDIO() call, or all lines at once.

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 *piChannelsArray* and *pbValueArray*

Returns:

TRUE if no error, FALSE otherwise (see p. 11)

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

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, for the possible options see PI_DRC()

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

Returns:

TRUE if no error, FALSE otherwise (see p. 11)

BOOL **PI_qDRR** (int *ID*, const int* *piRecTableIdArrays*, int *iNumberOfRecTables*, int *iOffset*, int *nrValues*, double** *pdValArray*, 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.

Notes:

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 general information regarding the data recording you can call PI_qHDR() which lists available record options and trigger options and gives additional information about data recording. The E-861 has 2 data recorder tables (ask with PI_qTNR()) with 1024 data points per table.

The data recorder configuration, i.e. the assignment of data sources and record options to the recorder tables, can be changed with PI_DRC(), and the current configuration can be read with PI_qDRC(). Data recorder tables with record option 0 are deactivated, i.e. nothing is recorded. Power-on default configuration: in data recorder table 1, the target position of axis 1 is recorded (record option 1) and in table 2 the current position of axis 1 (record option 2). For systems which have no position sensor, meaningless values are recorded with the default configuration.

Recording can be triggered in several ways. Ask with PI_qDRT() for the current trigger option and use PI_DRT() to change it. A trigger option set with PI_DRT() will become valid for all data recorder tables with non-zero record option. By default data recording is triggered with any command which changes position (move command).

The record table rate can be set with PI_RTR(). The power-on default of this value is one servo cycle (ask with PI_qRTR()). You can cover longer periods by increasing the record table rate.

Recording always takes place for all data recorder tables with non-zero record options.

Recording ends when the content of the data recorder tables has reached the maximum number of points.

Arguments:

ID ID of controller

piRecTableIdsArray IDs of data record tables

iNumberOfRecTables number of record 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_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.

Notes:

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 regarding data recording see the notes in **PI_qDRR()**.

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. 11)

BOOL PI_qDRT (int *ID*, const int* *piRecordTableIdsArray*, const char* *szTriggerSourceArray*, double* *pdValueArray*, int *iArraySize*)

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

szTriggerSourceArray array to receive the trigger source, see **PI_DRT()** for details.

pdValueArray array to receive the trigger-source-dependent value

iArraySize the size of the arrays *piRecordTableIdsArray* and *szTriggerSourceArray*

Returns:

TRUE if no error, **FALSE** otherwise (see p. 11)

BOOL PI_qERR (int *ID*, int* *piError*)

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. 71.

Arguments:

ID ID of controller

piError integer to receive error code of the controller

Returns:

TRUE if query successful, **FALSE** otherwise

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

Corresponding command: FRF?

Indicates whether the given axis is referenced or not.

Notes:

An axis is considered as "referenced" when the current position value is set to a known position. This is the case when PI_FRF(), PI_FNL() or PI_FPL() was successfully used or when the position was set directly with PI_POS().

Arguments:

ID ID of controller

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

piResult 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_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 regarding data recording see the notes in PI_qDRR().

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. 11)

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. 11)

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 "System Parameter Overview," beginning on p. 64, for an appropriate list of all parameters available for your controller.

Note:

The listed parameters can be changed and/or saved using the following commands:

PI_SPA() affects the parameter settings in the volatile memory (RAM).

PI_WPA() writes parameters settings from the RAM to the non-volatile memory.

PI_SEP() writes parameter settings directly into the non-volatile memory (without changing the RAM settings).

PI_RPA() resets the RAM to the values from the non-volatile memory.

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. 11)

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. 11)

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 (must be 1 for E-861, which supports only 1 joystick axis per controller)

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_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 also PI_JON()

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 deactivate, 1 for activate)

iArraySize size of arrays

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qLIM (nt 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, char * szName, char * szBuffer, const int maxlen)

Corresponding command: MAC?

Get available macros, or list contents of a specific macro. If *szName* is empty or **NULL**, all available macros are listed in *szBuffer*, separated with line-feed characters. Otherwise the content of the macro with name *szName* is listed, the single lines separated with by line-feed characters. If there are no macros stored or the requested macro is empty the answer will be "".

See "Working with Controller Macros" and the MAC command description in the controller User manual for details.

Arguments:

ID ID of controller

szName 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

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

Returns:

TRUE if successful, **FALSE** otherwise

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.

The target position can be changed by functions that cause motion (e.g. PI_MOV(), PI_MVR(), PI_GOH(), PI_STE()) or by the joystick (when disabling a joystick, the target position is set to the current position for joystick-controlled axes which are in 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 target positions of the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 11)

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

Corresponding **command**: OAC?

Get current open-loop acceleration of *szAxes*.

If all arguments are omitted, gets current value of all axes.

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. 11)

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

Corresponding **command**: ODC?

Get current open-loop deceleration of *szAxes*.

If all arguments are omitted, gets current value of all axes.

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 (long *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. 11)

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

Corresponding command: ONT?

Check if *szAxes* have reached the target.

The detection of the on-target status differs for closed-loop and open-loop operation.

Closed-loop operation:

The on-target status is influenced by two parameters: settle window (parameter ID 0x36) and settle time (parameter ID 0x3F).

The on-target status is true when the current position is inside the settle window and stays there for at least the settle time. The settle window is centered around the target position.

Open-loop operation:

The on-target status is true when the trajectory has finished which was internally calculated for the motion.

Arguments:

ID ID of controller

szAxes string with axes

pbValueArray array to be filled with current on-target status of the axes

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. 11)

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.

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. 11)

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

Corresponding command: RMC?

List macros which are currently running.

See "Working with 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 (const int *ID*, char *const *szAxes*, BOOL * *pbValarray*)

Corresponding command: RON?

Gets reference mode for given axes. See PI_RON() for a detailed description of reference mode.

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_qRTR (int *ID*, int *iRecordTableRate*)

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

iRecordTableRate variable to be filled with the record table rate

Returns:

TRUE if no error, **FALSE** otherwise (see p. 11)

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.

See also "Accessible Items and Their Identifiers" on p. 3.

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. 11)

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. 11)

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 number in the corresponding element of *iParameterArray*. See "System Parameter Overview," beginning on p. 64, for a list of all parameters.

You can query either all parameters or one single parameter per call of PI_qSEP().

The GEMAC parameters (ID 0x7000010 to ID 0x700001F) cannot be queried with PI_qSEP(). Use PI_qSPA() immediately after power-on or reboot to read the non-volatile values of these parameters. See "GEMAC Parameters" in the E-861 User manual for more information.

Arguments:

ID ID of controller

szAxes string with designator, one parameter is read for each designator!ID in szAxes; can be an axis or a PiezoWalk channel, the identifier is always 1.

piParameterArray parameter numbers

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. 11)

Errors:

PI_INVALID_SPP_CMD_ID one or more of the corresponding IDs in *iParameterArray* is invalid.

BOOL PI_qSPA (int *ID*, const char* *szAxes*, const int* *piParameterArray*, double* *pdValueArray*, char* *szStrings*, int *iMaximumStringSize*)

Corresponding command: SPA?

Query specified parameters for szAxes from RAM. For each desired parameter you must specify a designator in szAxes and the parameter number in the corresponding element of *iParameterArray*. See "System Parameter Overview," beginning on p. 64, for a list of all parameters.

You can query either all parameters or one single parameter per call of PI_qSPA().

Arguments:

ID ID of controller

szAxes string with designator, one parameter is read for each designator in *szAxes*; can be an axis or a PiezoWalk channel, the identifier is always 1.

piParameterArray parameter numbers

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)

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

Returns:

TRUE if no error, **FALSE** otherwise (see p. 11)

Errors:

PI_INVALID_SPP_CMD_ID one or more of the corresponding IDs in *iParameterArray* is invalid.

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, can be 1

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:

Bit	15	14	13	12	11	10	9	8
Description	On Target	Is referencing	Is Moving	Servo On	Transitions between motion and servo modes: Analog = 0 Relaxing = 1 Go to analog = 2 Relaxed = 3 In motion = 4 Go to motion = 5 See "Modes of Operation" in the E-861 User manual for details.			

Bit	7	6	5	4	3	2	1	0
Description	Digital Input 4	Digital Input 3	Digital Input 2	Digital Input 1	-	Positive Limit	Reference	Negative Limit

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. See PI_SSA() for details about the motion affected by the current voltage amplitude setting.

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. 11)

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. 11)

BOOL PI_qTAC (int *ID*, int * *pnNr*)

Corresponding command: TAC?

Get the number of installed analog channels.

Gets the number of analog input lines located on the I/O socket of the E-861 (Input 1 to Input 4). Note that these lines can also be used for digital input. See "Accessible Items and Their Identifiers" on p. 3 for more information.

Arguments:

ID ID of controller

pnNr pointer to *int* to receive the number of installed analog channels

Returns:

TRUE if successful, **FALSE** otherwise

BOOL PI_qTAV (int *ID*, const char* *szAnalogInputChannels*, int* *piValueArray*)

Corresponding command: TAV?

Returns voltage value for the specified analog input channel.

Using PI_qTAV(), you can directly read the Input 1 to Input 4 lines on the I/O socket of the controller. The identifiers of the lines are 1 to 4. See "Accessible Items and Their Identifiers" on p. 3 for more information. You can record the values of the analog input lines using the PI_DRC() record option 81.

Arguments:

ID ID of controller

szAnalogInputChannels string with channels. If "" or **NULL** all analog input channels are queried.

pdValueArray array to receive voltage value (in volts)

Returns:

TRUE if no error, **FALSE** otherwise (see p. 11)

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. 11)

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

Corresponding command: TMN?

Get the low end of the travel range of *szAxes*

The minimum commandable position is defined by the MAX_TRAVEL_RANGE_NEG parameter, ID 0x30.

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. 11)

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

Corresponding command: TMX?

Get the high end of the travel range of *szAxes*

The maximum commandable position is defined by the MAX_TRAVEL_RANGE_POS parameter, ID 0x15.

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. 11)

BOOL PI_qTNR (int *ID*, int* *piNumberOfRecordTables*)

Corresponding command: TNR?

Returns the number of data recorder tables.

The E-861 has 2 data recorder tables with 1024 data points per table.

For more information regarding data recording see the notes in PI_qDRR.

Arguments:

ID ID of controller

piNumberOfRecordTables variable to receive number of data recorder tables

Returns:

TRUE if no error, **FALSE** otherwise (see p. 11)

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, **FALSE** if not

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: VEL?

Get the current value of the closed-loop velocity for *szAxes*.

For the current open-loop velocity, query with PI_qOVL().

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. 11)

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. The aim of this procedure is to reduce all applied voltages when the target is reached and thus to increase the lifetime of the piezos.

Notes:

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

With the Relaxing procedure performed by PI_RNP(), the NEXACT® linear drive is brought to a full-holding-force, zero-drive-voltage Relaxed state.

PI_RNP() should reduce the voltages to zero without changing the position of the NEXACT® linear drive.

But depending on the current load, which may lead to a small motion of the movable part of the mechanics during the PI_RNP() procedure, small changes in position can occur.

In open-loop operation, PI_RNP() must be called each time the motion mode is to be changed from nanostepping motion (PI_OSM(), PI_OMA(), PI_OMR()) to analog motion (PI_OAD()) or vice versa.

After open-loop analog motion was done with PI_OAD(), PI_RNP() must be called before the servo can be switched on with PI_SVO() for closed-loop operation.

The PI_RNP() procedure can take up to four times the slewrate value (parameter ID 0x7000002; can be changed with PI_SPA() and PI_SEP()).

You can query the current state of the system (E-861 and NEXACT® linear drive) using PI_qSRG().

See "Modes of Operation" in the E-861 User manual for more information and Section 2.4 on p. 6 for an example.

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. 11)

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

Corresponding command: RON

Sets reference mode for given axes.

If the reference mode of an axis is ON, the axis must be driven to the reference switch (PI_FRF()) or, if no reference switch is available, to a limit switch (using PI_FPL(), PI_FNL()) before any other motion can be commanded.

If reference mode is OFF, no referencing is required for the axis. Only relative moves can be commanded (PI_MVR()), unless the actual position is set with PI_POS(). Afterwards, relative and absolute moves can be commanded.

For stages with neither reference nor limit switch, reference mode should be OFF.

WARNINGS:

If reference mode is switched off, and relative moves are commanded, stages can be driven into the mechanical hard stop if moving to a position which is outside the travel range!

If reference mode is switched off, and the actual position is incorrectly set with PI_POS(), stages can be driven into the mechanical hard stop when moving to a position which is thought to be within the travel range of the stage, but actually is not.

Arguments:

ID ID of controller

szAxes string with axes

pbValarray reference modes for the specified axes

Returns:

TRUE if successful, **FALSE** otherwise

Errors:

PI_CNTR_STAGE_HAS_NO_LIM_SWITCH if the axes have no reference or limit switches, and reference mode can not be switched ON

BOOL PI_RPA (int *ID*, const char* *szAxes*, const int* *piParameterArray*)

Corresponding command: RPA

Copy 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 number in the corresponding element of *piParameterArray*.

See "System Parameter Overview," beginning on p. 64, for a list of all parameters.

You can reset either all parameters or one single parameter with PI_RPA().

Arguments:

ID ID of controller

szAxes string with designators, one parameter is copied for each designator in *szAxes*; can be an axis or a PiezoWalk channel, the identifier is always 1.

piParameterArray parameter numbers

Returns:

TRUE if no error, **FALSE** otherwise (see p. 11)

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.

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. 11)

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 number in the corresponding element of *piParameterArray*. See "System Parameter Overview" beginning on p. 64, for a list of all parameters.

With the E-861, you can write only one single parameter per call of PI_SEP().

The GEMAC parameters (ID 0x7000010 to ID 0x700001F) can not be changed with PI_SEP(). Use PI_SPA() and PI_WPA() instead to save their values to non-volatile memory. See "GEMAC Parameter Adjustment" in the E-861 User manual for more information.

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`; can be an axis or a PiezoWalk channel, the identifier is always 1.

piParameterArray Parameter numbers

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. 11)

```
BOOL PI_SPA (int ID, const char* szAxes, const 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 number in the corresponding element of `iParameterArray`. See "System Parameter Overview," beginning on p. 64, for a list of all parameters.

Notes:

To save the currently valid parameters to non-volatile memory, where they become the power-on defaults, you must use `PI_WPA` (p. 58). Parameter changes not saved with `PI_WPA` will be lost when the controller is powered down.

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!

With the E-861, you can write only one single parameter per call of `PI_SPA()`.

The GEMAC parameters with the IDs 0x7000015, 0x7000016, 0x700001C, 0x700001D, 0x7000001E and 0x700001F are write-protected and can not be changed with `PI_SPA()`, even though the E-861 will send no error message if you try to change those parameters anyway.

Arguments:

ID ID of controller

szAxes string with designators, one parameter is set for each designator in `szAxes`; can be an axis or a PiezoWalk channel, the identifier is always 1.

piParameterArray Parameter numbers

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. 11)

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.

With the E-861, *pdValueArray* gives the feed voltage amplitude which can be 0 to 55 V for NEXACT® linear drives.

PI_SSA() changes the value of the Bending Voltage parameter (ID 0x07000003) in volatile memory (can be saved as power-on default with PI_WPA(), can also be changed with PI_SPA() and PI_SEP()).

The PI_SSA() setting takes effect for nanostepping motion in open-loop operation (servo off). Decreasing the step size will decrease the velocity for open-loop nanostepping motion. The velocity for open-loop nanostepping motion is also influenced by the open-loop velocity set with PI_OVL().

PI_SSA() can be used while the PiezoWalk channel is moving.

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. 11)

BOOL PI_STE (int *ID*, const char* *szAxes*, double* *pdStepSize*)

Corresponding command: *STE*

Starts performing a step and recording the step response for the given axis.

Caution: Do not use this command if no sensor is present in your system. Otherwise the connected mechanics can run into the hardstop at full speed which may cause damage to your hardware setup.

A "step" consists of a relative move of the specified amplitude which is performed relative to the current position. The axis must not be referenced before a measurement is performed.

The data recorder configuration, i.e. the assignment of data sources and record options to the recorder tables, can be set with PI_DRC().

The recorded data can be read with PI_DRR().

For more information regarding data recording see the notes in PI_qDRR().

Step response measurements provide meaningful results only in closed-loop operation.

Motion functions like PI_STE() are not allowed when the joystick is active for the axis. See "Joystick Control" in the E-861 User manual for details.

Arguments:

ID ID of controller

szAxes axes for which the step response will be recorded

pdStepSize size of step (amplitude)

Returns:

TRUE if no error, FALSE otherwise (see p. 11)

BOOL PI_STP (int *ID*)

Corresponding command: *STP*

Stops the motion of all axes instantaneously. Sets error code to 10.

After the stage was stopped, if servo is on the target position is set to the current position.

PI_STP() stops all motion caused by move functions (PI_MOV(), PI_MVR(), PI_OAD(), PI_OSM(), PI_OMA(), PI_OMR(), PI_STE()), referencing commands (PI_FNL(), PI_FPL(), PI_FRF()) and macros (PI_MAC()).

Arguments:

ID ID of controller

Returns:

TRUE if successful, FALSE otherwise (see p. 11)

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

Corresponding command: SVO

Set servo-control "on" or "off" (closed-loop/open-loop mode). If *pbValueArray[index]* is **FALSE** the mode is "off", if **TRUE** it is set to "on".

Caution: If no sensor is present in your system, do not switch the servo on with PI_SVO() command and do not send commands that require a sensor, like PI_MOV(), PI_MVR(), PI_OMA() or PI_OMR(). Otherwise the connected mechanics can run into the hardstop at full speed which may cause damage to your hardware setup.

When switching from open-loop to closed-loop operation, the target is set to the current position to avoid jumps of the mechanics.

Switching the servo off includes an automatic Relaxing procedure which brings the NEXACT® linear drive to a full-holding-force, zero-drive-voltage Relaxed state. Switching the servo on includes preparation of a "relaxed" NEXACT® linear drive for closed-loop motion. These actions can take up to four times the slewrate value (parameter ID 0x7000002). See "Modes of Operation" in the E-861 User manual for more information.

The current servo state affects the applicable move functions:

servo-control off: use PI_OSM(), PI_OMA(), PI_OMR() and PI_OAD()

servo-control on: use PI_MOV() and PI_MVR()

When servo is switched off while the axis is moving, the axis stops. Exception: While the axis is under joystick control (a joystick connected to the E-861 is enabled with PI_JON()), motion will continue when the servo is switched on or off for the axis.

Using a startup macro, you can configure the controller so that servo is automatically switched on upon power-on or reboot. See "Start-Up Macro" in the E-861 User manual for details.

Arguments:

ID ID of controller

szAxes string with axes

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

Returns:

TRUE if no error, **FALSE** otherwise (see p. 11)

BOOL PI_VEL (int *ID*, const char* *szAxes*, 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).

The lowest possible value for *pdValueArray* is 0.

PI_VEL() changes the value of the Current closed-loop velocity parameter (ID 0x49) in volatile memory (can be saved as power-on default with PI_WPA(), can also be changed with PI_SPA() and PI_SEP()).

The maximum value which can be set with PI_VEL() given by the Maximum closed-loop velocity parameter, ID 0xA (can be changed with PI_SPA() and PI_SEP()).

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray velocities for the axes

Returns:

TRUE if no error, **FALSE** otherwise (see p. 11)

BOOL PI_WAC (int *ID*, 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 to a specified rule.

Can only be used in macros.

See also PI_MEX()

See "Working with Controller Macros" and the WAC command description in the controller User manual for details and examples.

Arguments:

ID ID of controller

szCondition string with condition to evaluate

Returns:

TRUE if successful, **FALSE** otherwise (see p. 11)

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 number in the corresponding element of *piParameterArray*. See "System Parameter Overview" beginning on p. 64 for a list of all parameters.

Notes:

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.

With PI_qHPA() you can obtain a list of the parameters IDs.

Use PI_qSPA() to check the current parameter settings in the volatile memory.

Warning: The number of write times of non-volatile memory is limited. Do not write parameter values except when necessary.

Parameters can be changed in volatile memory with PI_SPA() and PI_SSA().

When PI_WPA() is used without specifying any arguments except of the password, the currently valid values of all parameters affected by the specified password are saved. Otherwise only one single parameter can be saved per call of PI_WPA().

Arguments:

ID ID of controller

szPassWord The password for writing to non-volatile memory depends on the parameter and can be "100" or "4711". See the parameter list in "System Parameter Overview" on p. 64 for the password assignment.

szAxes string with designators. For each designator in *szAxes* one parameter value is copied; can be an axis or a PiezoWalk channel, the identifier is always 1.

piParameterArray Array with parameter numbers

Returns:

TRUE if no error, **FALSE** otherwise (see p. 11)

7. Functions for User-Defined Stages

7.1. Overview

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 (see “Parameter 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 “System Parameter Overview” on p. 64 for a complete 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 addition 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.

If an older version of the software was installed an existing PI_UserStages.dat is automatically converted into PI_UserStages2.dat. For details see “Parameter Databases” below.

```

BOOL PI_FUNC_DECL PI_AddStage (const int ID, char *const szAxes)
BOOL PI_FUNC_DECL PI_RemoveStage (const int ID, char *szStageName)
BOOL PI_FUNC_DECL PI_OpenUserStagesEditDialog (const int ID)
BOOL PI_FUNC_DECL PI_OpenPiStagesEditDialog (const int ID)

```

7.2. Function Description

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

Adds the stage specified for *szAxes* to the file *PI_UserStages2.dat* with the 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_OpenPiStagesEditDialog (const int *ID*)

Opens a dialog to look at the *PiStages2.dat* file, which contains the stages defined by PI. No changes can be made to this file.

CAUTION: This function is provided for compatibility reasons only. It is not recommended to open the *PIStageEditor* this way. 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 *PIMicroMove* or via its executable to check stage parameter sets in *PiStages2.dat*.

Arguments:

ID ID of controller

Returns:

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

BOOL PI_OpenUserStagesEditDialog (const int *ID*)

Opens a dialog to edit, add and remove stages from the *PI_UserStages2.dat* file, which contains the user-defined stages.

CAUTION: This function is provided for compatibility reasons only. It is not recommended to open the *PIStageEditor* this way. 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 *PIMicroMove* or via its executable to edit, add or remove stage parameter sets in *PI_UserStages2.dat*.

Arguments:

ID ID of controller

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 the 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

7.3. Parameter Databases

The PI GCS 2 DLL and the GCS-based host software from PI use multiple databases for stage parameters:

- **PIStages2.dat** contains parameter sets for all standard stages from PI and is automatically installed on the host PC with the setup. It cannot be edited; should changes in the file become necessary, you must obtain a new version from PI and install it on your host PC (see “Updating PIStages2.dat”, p. 62).
- **PI_UserStages2.dat** allows you to create and save your own stages (see “Overview” on p. 59). This database 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 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).
- **N-xxx.dat** files contain parameter sets for custom stages delivered by PI. Those files are provided with the stages and have to be copied to the host PC according to the accompanying instructions.
N-xxx.dat files cannot be edited; should changes become necessary, you must obtain a new version from PI.

The PIStages2.dat, PI_UserStages2.dat and N-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). 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.

Notes for users which have already installed older versions of PI GCS 2 DLL, PIMikroMove and PiStageEditor:

- The format of the stage parameter (DAT) files has changed (more parameters provided), realized by a file version change from 1 to 2. Note that PIStages and PIUserstages DAT files with version 2 contain a "2" in their file name, e.g. PIStages2.dat (instead of PIStages.dat for version 1).
- Existing PIUserstages 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. The *Edit user stages data...* item in the controller menu of PIMikroMove opens the PiStageEditor tool with the version 2 file (PIUserStages2.dat). Parameters which were not present in version 1 are set to default values during conversion.
- Version 4 and newer of the PiStageEditor supports stage parameter files of version 2 (in PIMikroMove, you can check the version of the PiStageEditor with *Help → About PiStageEditor*). If it is necessary to update the PiStageEditor, run either the setup from the latest revision of the CD for your controller, or download the latest revision of the PiStageEditor from the PI website. It can be found there in the same directory like the PIStages2.dat stage database. See “Updating PIStages2.dat” below for download instructions and make sure to copy the PiStageEditor.dll to the ...\\PI\\GcsTranslator directory.

7.4. Updating PIStages2.dat

To install the latest version of PIStages2.dat from the PI Website proceed as follows:

1. On the www.pi.ws front page, move the cursor to *Manuals, Software, ISO Statements* in the *Service* section on the left.
 2. Select *Software* from the list that pops up.
 3. On the *PI Support Site* page, click on the *General Software* category (no login or password is required).
 4. Click on *PI Stages*.
 5. Click on *pistages2*.
 6. In the download window, switch to the `...\PI\GcsTranslator` directory. 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)` (may differ in other-language Windows versions).
- Note that in PIMikroMove, you can use the *Version Info* entry in the controller menu or the *Search for controller software* entry in the *Connections* menu to identify the GcsTranslator path or open the directory in the explorer using the *Show GCS Path* button.
7. If desired, rename the existing PIStages2.dat (if present) so as to preserve a copy for safety reasons.
 8. Download the file from the server as PIStages2.dat.

7.5. Troubleshooting

Problem:

Stage DAT file can not 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 *PIStages2.dat* and *PI_UserStages2.dat* files. Although PI aims for highest compatibility, the latest host software may not be able to work with older stage DAT files. You can check the revision of you stage DAT 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 download and replacement instructions see "Updating PIStages2.dat" above.

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 can not 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. See the “System Parameter Overview” Section (p. 64) for details

8. System Parameter Overview

To adapt the E-861 to your application, you can modify parameter values. The parameters available depend on the controller firmware. With PI_qHPA() you can obtain a list of all available parameters with information about each (e.g. short descriptions). The volatile and non-volatile memory parameter values can be read with the PI_qSPA() or PI_qSEP() functions, respectively.

Using the "general" modification functions PI_SPA(), PI_RPA(), PI_SEP() and PI_WPA(), parameters can be changed in volatile memory (PI_SPA(), PI_RPA()) or in non-volatile memory (PI_SEP(), PI_WPA()). It is recommended that any modifications be first made with PI_SPA(), and when the controller runs well, saved using PI_WPA(). In addition to the "general" modification functions, there are functions which change certain specific parameters in volatile memory (see table below).

The values of the parameters required for closed-loop operation depend on your mechanics (stage configuration, sensor, load) and have to be adjusted properly before initial operation of a closed-loop system. See "Customizing the System" in the E-861 User manual for detailed information.

For detailed information regarding the parameters for the GEMAC interpolation circuit (IDs 0x7000010 to 0x700001F), see "GEMAC Parameters" in the E-861 User manual and the GEMAC manual IP1000_B_EN.pdf which is provided on the E-861 CD.

CAUTION

Incorrect E-861 parameter values may lead to improper operation or damage of your hardware. Be careful when changing parameters.

Values stored in non-volatile memory are power-on defaults, so that the system can be used in the desired way immediately.

Parameter ID (hexa-decimal)	Data Type	Password for Writing to Non-Volatile Memory	Parameter Description	Possible Values/Notes
0x1	INT	100	P-Term of PID-parameter set for nanostepping mode	0 to 65535
0x2	INT	100	I-Term of PID-parameter set for nanostepping mode	0 to 65535
0x3	INT	100	D-Term of PID-parameter set for nanostepping mode	0 to 65535
0x4	LONG INT	100	I-limit	0 to 2,000,000

Parameter ID (hexa-decimal)	Data Type	Password for Writing to Non-Volatile Memory	Parameter Description	Possible Values/Notes
0x8	FLOAT	100	Maximum position error (user unit)	Used for stall detection. If the position error (i.e. the absolute value of the difference between current position and commanded position) in closed-loop operation exceeds the given maximum, the controller sets error code -1024 ("Motion error"), the servo will be switched off and the axis stops. Note that with open-loop referencing exceeding of maximum position error is not indicated.
0xA	FLOAT	100	Closed-loop velocity (maximum; user unit/s)	Gives the maximum value which can be set with PI_VEL(). On power-on, the current closed-loop velocity is set by parameter 0x4B.
0xB	FLOAT	100	Closed-loop acceleration (user unit/s ²)	
0xC	FLOAT	100	Closed-loop deceleration (user unit/s ²)	
0xE	INT	100	Numerator of the counts-per-physical-unit factor	1 to 1,000,000 for each parameter. The counts-per-physical-unit factor determines the "user" unit for closed-loop motion commands. When you change this factor, all other parameters whose unit is based on the "user" unit are adapted automatically, e.g. closed-loop velocity and parameters regarding the travel range.
0xF	INT	100	Denominator of the counts-per-physical-unit factor	
0x14	INT	100	Stage has a reference	1 = the stage has a reference, else 0
0x15	FLOAT	100	MAX_TRAVEL_RANGE_POS The maximum travel in positive direction (user unit)	"Soft limit", based on the home (zero) position. If the soft limit is smaller than the position value for the positive limit switch (which is given by the sum of the parameters 0x16 and 0x2F), the positive limit switch cannot be used for referencing. Smallest possible value is 0.
0x16	FLOAT	100	VALUE_AT_REF_POS The position value at the reference position (user unit)	The position value which is to be set when the mechanics performs a reference move to the reference switch.

Parameter ID (hexa-decimal)	Data Type	Password for Writing to Non-Volatile Memory	Parameter Description	Possible Values/Notes
0x17	FLOAT	100	DISTANCE_REF_TO_N_LIM The distance between reference switch and negative limit switch (user unit)	Represents the physical distance between the reference switch and the negative limit switch integrated in the mechanics. When the mechanics performs a reference move to the negative limit switch, the position is set to the difference of VALUE_AT_REF_POS and DISTANCE_REF_TO_N_LIM.
0x18	INT	100	Axis limit mode	0 = positive limit switch active high (pos-HI), negative limit switch active high (neg-HI) 1 = positive limit switch active low (pos-LO), neg-HI 2 = pos-HI, neg-LO 3 = pos-LO, neg-LO
0x2F		100	DISTANCE_REF_TO_P_LIM The distance between reference switch and positive limit switch (user unit)	Represents the physical distance between the reference switch and the positive limit switch integrated in the mechanics. When the mechanics performs a reference move to the positive limit switch, the position is set to the sum of VALUE_AT_REF_POS and DISTANCE_REF_TO_P_LIM.
0x30	FLOAT	100	MAX_TRAVEL_RANGE_NEG The maximum travel in negative direction (user unit)	"Soft limit", based on the home (zero) position. If the soft limit is smaller than the absolute value of the position set for the negative limit switch (i.e. the difference of the parameters 0x16 and 0x17), the negative limit switch cannot be used for referencing. Smallest possible value is 0.
0x31	INT	100	Invert the reference	1 = invert the reference, else 0
0x32	INT	100	Stage has limit switches; enables / disables the stopping of the motion at the limit switches	0 = Stage has limit switches 1 = Stage has no limit switches
0x36	INT	100	Settle window (counts)	In closed-loop operation, the on-target status is true when the current position is inside the settle window and stays there for at least the settle time (parameter ID 0x3F). The settle window is centered around the target position. The minimum value for stable positioning is 2.
0x3C	STRING	100	Stage name	Default is "DEFAULT_STAGE-N"
0x3F	FLOAT	100	Settle time (ms)	0 to 1000 ms; see Settle window
0x47	INT	100	Default direction for reference	0 = detect automatically, 1 = start in negative direction, 2 = start in positive direction

Parameter ID (hexa-decimal)	Data Type	Password for Writing to Non-Volatile Memory	Parameter Description	Possible Values/Notes
0x49	FLOAT	100	Current closed-loop velocity (user unit/s)	Gives the current closed-loop velocity limited by parameter 0xA
0x4A	FLOAT	100	Maximum closed-loop acceleration (user unit/s ²)	Gives the maximum value which can be set with PI_ACC (p. 25) (for parameter 0xB)
0x4B	FLOAT	100	Maximum closed-loop deceleration (user unit/s ²)	Gives the maximum value which can be set with PI_DEC (p. 25) (for parameter 0xC)
0x50	FLOAT	100	Closed-loop referencing velocity (user unit/s)	Is limited by Maximum closed-loop velocity parameter (ID 0xA)
0x5A	INT	100	Numerator of servo loop input factor	<p>1 to 1,000,000 for each parameter.</p> <p>The servo loop input factor decouples servo loop parameters from the encoder resolution.</p> <p>Not to be changed by the customer.</p> <p>Note that the servo loop input factor is independent from 0xE and 0xF, i.e., the counts-per-physical-unit factor.</p> <p>The counts-per-physical-unit factor has no influence on control loop stability, but is used for input and output scaling of position values</p>
0x5B	INT	100	Denominator of servo loop input factor	
0x94	FLOAT	100	Notch filter frequency 1 (Hz)	<p>40 to 20000</p> <p>The corresponding frequency component in the control value is reduced to compensate for unwanted resonances in the mechanics.</p> <p>Only active in closed-loop operation. Should normally not be changed (try to change only with very high loads).</p>
0x95	FLOAT	100	Notch filter edge	<p>0.1 to 10</p> <p>Gives the slope of the filter edge. Do not change</p>
0xAC	FLOAT	100	Low-pass filter frequency (Hz)	<p>40 to 20000</p> <p>Gives the cut-off frequency of a low-pass filter which is only active in closed-loop operation and with analog mode motion</p>
0x401	INT	100	P term of PID-parameter set for analog mode	0 to 65535
0x402	INT	100	I term of PID-parameter set for analog mode	0 to 65535
0x403	INT	100	D term of PID-parameter set for analog mode	0 to 65535

Parameter ID (hexa-decimal)	Data Type	Password for Writing to Non-Volatile Memory	Parameter Description	Possible Values/Notes
0x7000000	INT	100	Travel range minimum (user unit)	By default set to a very large value which should not be changed. Can be used as travel range limit for open-loop motion with PI_OSM (p. 37), PI_OSMf (p. 38), PI_OAD (p. 35), PI_OMA(p. 36), PI_OMR (p. 36): If the current position reaches this value, the motion is stopped. In this case, enlarge the limit.
0x7000001	INT	100	Travel range maximum (user unit)	By default set to a very large value which should not be changed. Can be used as travel range limit for open-loop motion with PI_OSM (p. 37), PI_OSMf (p. 38), PI_OAD (p. 35), PI_OMA(p. 36), PI_OMR (p. 36): If the current position reaches this value, the motion is stopped. In this case, enlarge the limit.
0x7000002	INT	100	Slewrates (ms)	1 to 1000 ms This value affects the time which is required for the "preparing for motion" and "relaxing" changes of the motion state. Depending on the current motion state, a change of state can take up to four times the slewrates value. See "Modes of Operation" in the E-861 User manual for details.
0x7000003	FLOAT	100	Bending voltage, also changed by PI_SSA()	0 to 55 V, gives the step size for open-loop nanostepping motion
0x7000004	INT	100	Time to close servo (ms)	1 to 1000 ms Gives the time interval between obtaining trajectory voltage and closing of servo loop
0x7000201	INT	100	Current open-loop velocity (step cycles/s)	Gives the current open-loop velocity, limited by parameter 0x7000204, can be set with PI_OVL() (p. 39). On power-on, the current open-loop velocity is set by parameter 0x7000201.
0x7000202	FLOAT	100	Current open-loop acceleration (step cycles/s ²)	Gives the current open-loop acceleration, limited by parameter 0x7000205, can be set with PI_ACC()(p. 25)
0x7000203	INT	100	Current open-loop deceleration (step cycles/s ²)	Gives the current open-loop deceleration, limited by parameter 0x7000206, can be set with PI_DEC()(p. 25)
0x7000204	FLOAT	100	Maximum open-loop velocity (step cycles/s)	Gives the maximum value which can be set with PI_OVL().

Parameter ID (hexa-decimal)	Data Type	Password for Writing to Non-Volatile Memory	Parameter Description	Possible Values/Notes
0x7000205	FLOAT	100	Maximum open-loop acceleration (step cycles/s ²)	Gives the maximum value which can be set for parameter 0x7000202
0x7000206	FLOAT	100	Maximum open-loop deceleration (step cycles/s ²)	Gives the maximum value which can be set for parameter 0x7000203
0x7000207	INT	100	Open-loop referencing velocity (step cycles/s)	Is limited by Maximum open-loop velocity parameter (ID 0x7000204)
0x7001A00	INT	100	PiezoWalk Driving mode	Determines how closed-loop motion is performed: 0 = nanostepping mode only 1 = alternating sequence of analog mode and nanostepping mode
0x700601	CHAR		Axis unit	This parameter has to be set consistent with the counts-per-physical-units factor given by parameters 0xE and 0xF
0x7000010		4711	GEMAC parameter 0 (Write register CFG0 on the GEMAC interpolation circuit)	Can be set with PI_SPA() and saved with PI_WPA(). Not available for PI_SEP() and PI_SEP()
0x7000011		4711	GEMAC parameter 1 (Write register CFG1 on the GEMAC interpolation circuit)	Can be set with PI_SPA() and saved with PI_WPA(). Not available for PI_SEP() and PI_SEP()
0x7000012		4711	GEMAC parameter 2 (Write register CFG2 on the GEMAC interpolation circuit)	Can be set with PI_SPA() and saved with PI_WPA(). Not available for PI_SEP() and PI_SEP()?
0x7000013		4711	GEMAC parameter 3 (Write register ERRMASK on the GEMAC interpolation circuit)	Can be set with PI_SPA() and saved with PI_WPA(). Not available for PI_SEP() and PI_SEP()
0x7000014		4711	GEMAC parameter 4 (Write register PHASE on the GEMAC interpolation circuit)	Can be set with PI_SPA() and saved with PI_WPA(). Not available for PI_SEP() and PI_SEP()
0x7000015		-	GEMAC parameter 5 (reserved)	Read only. Not included in PI_SEP() response
0x7000016		-	GEMAC parameter 6 (reserved)	Read only. Not included in SEP? response
0x7000017		4711	GEMAC parameter 7 (Write register SGAIN on the GEMAC interpolation circuit)	Can be set with PI_SPA() and saved with PI_WPA(). Not available for PI_SEP() and PI_SEP()
0x7000018		4711	GEMAC parameter 8 (Write register SOFF on the GEMAC interpolation circuit)	Can be set with PI_SPA() and saved with PI_WPA(). Not available for PI_SEP() and PI_SEP()

Parameter ID (hexa-decimal)	Data Type	Password for Writing to Non-Volatile Memory	Parameter Description	Possible Values/Notes
0x7000019		4711	GEMAC parameter 9 (Write register CGAIN on the GEMAC interpolation circuit)	Can be set with PI_SPA() and saved with PI_WPA(). Not available for PI_SEP() and PI_SEP()
0x700001A		4711	GEMAC parameter 10 (Write register COFF on the GEMAC interpolation circuit)	Can be set with PI_SPA() and saved with PI_WPA(). Not available for PI_SEP() and PI_SEP()
0x700001B		4711	GEMAC parameter 11 (Write register SYNC on the GEMAC interpolation circuit)	Can be set with PI_SPA() and saved with PI_WPA(). Not available for PI_SEP() and PI_SEP()
0x700001C		-	GEMAC parameter 12 (internal use)	Read only. Not included in PI_SEP() response
0x700001D		-	GEMAC parameter 13 (internal use)	Read only. Not included in PI_SEP() response
0x700001E		-	GEMAC parameter 14 (internal use)	Read only. Not included in PI_SEP() response
0x700001F		-	GEMAC parameter 15 (internal use)	Read only. Not included in PI_SEP() response

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 can not 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 non-volatile 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 can not be started
69	PI_CNTR_DDL_TIME_DELAY_TOO_LARGE	Time delay larger than DDL table; DDL can not 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 did 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
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
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
555	PI_CNTR_UNKNOWN_ERROR	BasMac: unknown controller error
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
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)
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
5200	PI_CNTR_AXIS_NOT_CONFIGURED	Axis must be configured for this action

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!)

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.

10. Index

#5 26
#7 25
*IDN? 38
axis parameters 11
BOOL 12
boolean values 12
c strings 12
CST 20
CST? 33
CSV? 34
DEL 20
DIO 21
DIO? 34
DRC 21
DRC? 34
DRR? 35
DRT 22
DRT? 36
dynamic loading of a DLL 10
ERR? 36
Error codes 62
FALSE 12
FED 22
FNL 23
FPL 23
FRF 24
FRF? 36
GetProcAddress - Win32 API function 11
GOH 25
HLP? 37
HLT 25
HPA? 37
interface settings 17
JAS? 38
JAX 26
JAX? 38
JBS? 39
JDT 27
JON 27
JON? 39
LIB - static import library 10
LIM? 39
linking a DLL 10
LoadLibrary - Win32 API function 11
MAC BEG 27
MAC DEF 28
MAC DEF? 29
MAC DEL 28
MAC END 28
MAC START 29
MAC? 40
MEX 30
module definition file 10
MOV 30
MOV? 40
MVR 30
NULL 12
OAD 31
OAD? 40
ONT? 41
OSM 32
OVL 32
OVL? 41
PI_EnumerateUSB 15
PI_CloseConnection 14
PI_CloseDaisyChain 14
PI_ConnectDaisyChainDevice 14
PI_ConnectRS232 14
PI_ConnectUSB 14
PI_ConnectUSBWithBaudRate 15
PI_CST 20
PI_DEL 20
PI_DIO 21
PI_DRC 21
PI_DRT 22
PI_END 28
PI_FED 22
PI_FNL 23
PI_FPL 23
PI_FRF 24
PI_GetAsyncBuffer 24
PI_GetAsyncBufferIndex 25
PI_GetError 15
PI_GOH 25
PI_HLT 25
PI_InterfaceSetupDlg 15
PI_IsConnected 16
PI_IsControllerReady 25
PI_IsMoving 26
PI_IsRunningMacro 26
PI_JAX 26
PI_JDT 27
PI_JON 27
PI_MAC_BEG 27
PI_MAC_DEF 28
PI_MAC_DEL 28
PI_MAC_NSTART 29
PI_MAC_qDEF 29
PI_MAC_START 29
PI_MEX 30
PI_MOV 30
PI_MVR 30
PI_OAD 31
PI_OpenRS232DaisyChain 16
PI_OSM 32
PI_OVL 32
PI_POS 33
PI_qCST 33
PI_qCSV 34
PI_qDIO 34
PI_qDRC 34
PI_qDRR 35
PI_qDRR_SYNC 35
PI_qDRT 36

PI_qERR	36	PI_SVO	52
PI_qFRF	36	PI_TranslateError	17
PI_qHLP	37	PI_VEL	53
PI_qHPA	37	PI_WAC	53
PI_qIDN	38	PI_WPA	53
PI_qJAS	38	POS	33
PI_qJAX	38	POS?	41
PI_qJBS	39	RBT	48
PI_qJON	39	RMC?	42
PI_qLIM	39	RNP	48
PI_qMAC	40	RON	48
PI_qMOV	40	RON?	42
PI_qOAD	40	RPA	49
PI_qONT	41	RS-232 settings	17
PI_qOVL	41	RTR	49
PI_qPOS	41	RTR?	42
PI_qRMC	42	SAI?	42, 43
PI_qRON	42	SEP	49
PI_qRTR	42	SEP?	43
PI_qSAI	42, 43	settings for RS-232	17
PI_qSEP	43	SPA	50
PI_qSPA	44	SPA?	44
PI_qSRG	44	SRG?	44
PI_qSSA	45	SSA	51
PI_qSVO	45	SSA?	45
PI_qTAC	45	static import library	10
PI_qTAV	46	STE	51
PI_qTIO	46	STP	52
PI_qTMN	46	SVO	52
PI_qTMX	46	SVO?	45
PI_qTNR	47	TAC?	45
PI_qTRS	47	TAV?	46
PI_qVEL	47	TIO?	46
PI_RBT	48	TMN?	46
PI_RNP	48	TMX?	46
PI_RON	48	TNR?	47
PI_RPA	49	TRS?	47
PI_RTR	49	TRUE	12
PI_SEP	49	User-defined stages	55
PI_SetErrorCheck	16	VEL	53
PI_SPA	50	VEL?	47
PI_SSA	51	WAC	53
PI_STE	51	WPA	53
PI_STP	52		

