Piezo · Nano · Positioning

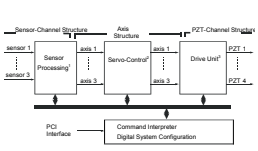
PZ163E Software Manual

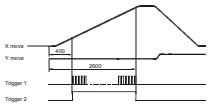
E-761 GCS Library (E7XX_GCS_DLL)

Release: 1.1.0 Date: 11 April 2008

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

- E-761.3xx 3-Channel Digital Piezo Controller PCI Board





Auf der Römerstr. 1 · 76228 Karlsruhe, Germany
Tel. +49-721-4846-0 · Fax: +49-721-4846-299
info@pi.ws · www.pi.ws

First printing 11 April 2008
Document Number PZ163E, BRo, Release 1.1.0
E-761_GCSDLL_PZ163E.doc

Table of Contents

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 to E7XX_GCS Library

The E-7xx_GCS library allows controlling one or more PI E-7xx 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** 2000, XP and Vista: E7XX_GCS_DLL
  See Section 2.2 starting on p. 6 for more information about PI DLLs.
  Windows Vista: The E-761 host software must always be started with the "Run as administrator" option. To do this, click on the Start menu entry or the executable file of the appropriate program with the right mouse button and select the "Run as administrator" entry from the context menu.
- **Linux** operating systems (kernel 2.6, GTK 2.0, glibc 2.4): libpi_e7xx_gcs.so.*x.x.x* and libpi_e7xx_gcs-*x.x.x*.a where *x.x.x* gives the version of the library

NOTE

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

1.1. Quick Start

1.1.1. Installation

NOTE

Users who have already installed E-761 hardware driver and software:
With release 2.0.0 or newer of the E-761 CD, a new hardware driver for the E-761 board is provided ("PI E761 Driver"). When using this driver, revision 4.0 or newer of the GCS library must be installed. You should run the installation procedure as described below to make sure that all components are updated.

Windows operating systems:

To install the E7XX_GCS_DLL on your host PC, proceed as follows:

- Be sure to login as administrator and insert the product CD in your host PC.
- If the Setup Wizard does not open automatically, start it from the root directory of the CD with the  icon.
  Note for Windows Vista: If the Setup Wizard starts automatically, cancel it. In the root directory of the CD, click on the setup.exe file with the right mouse button and select the "Run as administrator" entry from the context menu.
- Follow the on-screen instructions. You can choose between “typical” and “custom” installation. Typical components are hardware driver, LabView drivers, GCS DLL, NanoCapture™, PZTControl™, the Firmware Update Wizard and the manuals. “Typical” is recommended.

Linux operating systems:

1. Insert the E-761 CD in the host PC.
2. Open a terminal and go to the /linux directory on the E-761 CD.
3. Log in as superuser (root).
4. Start the install script with ./INSTALL
   Keep in mind the case sensitivity of Linux when typing the command.
5. Follow the on-screen instructions. You can choose the individual components to install.

If the installation fails, make sure you have installed the kernel header files for your kernel.

1.1.2. Connect the Controller

Physically connect the controller(s) to the PC. Multiple boards can be installed in one PC. Install the E-761 board(s) and the hardware driver in the PC as described in the E-761 User Manual.

To enable communication, use the DLL functions described in Section "Communication Initialization" on p. 11 and see also the examples given in Section "Controller Setup" on p. 9.

1.1.3. Examples

The sample program *E761QuickTest.exe* and the appropriate source code are to be found in the \Sample\c directory of the product CD.

PZTControl makes it possible to test the DLL functions in a convenient way (Windows operating systems only).

1.2. Units and GCS—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 x 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 x 10^{-6} working units followed by 1 move of 20 x 10^{-6} in the other direction cause a net motion of 1 hardware unit forward.

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

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

1.3. Stages, Axes, and Channels

The digital E-7xx controllers have the advantage that sensor and output channels can be combined in a flexibly programmable internal coordinate transformation. This means that the sensors and actuators geometry is independent of the logical coordinate system used for programming.

If PI had sufficient knowledge of your application, your system will be configured and calibrated at the factory before shipment.

NOTE

If you need to change axis definitions or certain other configuration values, it may be more convenient to use the *NanoCapture*™ software or a terminal emulator than to attempt to call the DLL functions from your own program. See the *NanoCapture*™ software manual for details on its convenient GUI.

1.3.1. Terminology

The terms “axis,” “channel” and “stage” are defined as follows:

- A *piezo (PZT) channel* is the representation of a PZT amplifier in the firmware. Multiple PZT amplifiers can be involved in the motion of one logical axis.
- A *sensor channel* is the representation of a physical existing sensor in the firmware. Multiple sensor channels can be involved in the control (measuring) of one logical axis.
- The user/programmer can monitor and command the stage motion based on a system of *logical axes*.
- A *stage* contains at least one piezo actuator, and may also contain at least one sensor. Each piezo actuator is connected to one PZT channel, and each sensor is connected to one sensor channel of the controller.

The PZT channels, sensor channels and logical axes need not coincide with one other or even be parallel. An E-7xx controller always uses the values in its coordinate transformation matrices to transform sensor data into the axis coordinate system and, when motion is required, to transform the axis information into PZT channel values. See the E-7xx User Manual for details.

Every effort has been made to use the terms described here consistently in this and other documentation.

1.3.2. Typical System Configurations

- Axes and channels correspond to one another. Example:
  - Independent single-axis stages. The axis names are arbitrary and can be assigned by the user.
- The number of axes may be different from the number of channels, i.e. each PZT and sensor channel can participate in more than one axis, and each axis can be driven by more than one PZT channel and measured by more than one sensor channel. Examples:
  - A rotation axis driven by a pair of PZT channels and monitored by a pair of sensors.
  - Two rotation axes and one linear axis, all driven by 3 PZT channels and monitored by 3 sensors.
  - One 3-axis stage with 4 piezo actuators and 3 sensors

1.3.3. Axis Renaming

The GCS DLL supports an axis-renaming scheme. See the **E7XX_SAI** function (p. 33) for details.

Keep in mind the following when dealing with axis names: Do not mistake the axis identifiers set with **E7XX_SAI** with the Axis name parameter (ID 0x07000600) which is only used in the graphical user interface of NanoCapture™. Valid axis names (identifiers) consist of only one character.

2. 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.

2.1. 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.

2.2. 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 `E7XX_GCS_DLL.DLL` in Delphi projects. Please see http://www.drbob42.com/delphi/headconv.htm for a detailed description of the steps necessary.)

2.2.1. Using a Static Import Library

The `E7XX_GCS_DLL.DLL` module is accompanied by the `E7XX_GCS_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.

2.2.2. 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
   E7XX_GCS_DLL.E7XX_IsConnected
```

2.2.3. 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_E7XX_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_E7XX_IsConnected)( int );
FP_E7XX_IsConnected pE7XX_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 us by the GetProcAddress() function. Example:

```
HINSTANCE hPI_Dll = LoadLibrary("E7XX_GCS_DLL.DLL\0");
```

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

```
pE7XX_IsConnected  = (FP_E7XX_IsConnected)GetProcAddress(hPI_Dll,"E7XX_IsConnected\0");
if (pE7XX_IsConnected == NULL)
{
    // do something, for example
    return FALSE;
}
BOOL bResult = (*pE7XX_IsConnected)(1); // call E7XX_IsConnected(1)
```

2.3. Function Calls

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

2.3.1. Error Return

Almost all functions will return a boolean value of type BOOL (see “Boolean Values” (p. 8)). The result will be non-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 **E7XX_SetErrorCheck**, p. 12) 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 **E7XX_GetError**()(p.12)) to obtain the error code, and, if desired, translate it to the corresponding error message with **E7XX_TranslateError** (p. 12). The error codes and messages are listed in “Error Codes” (p. 52).

2.3.2. Axis Identifiers

Many commands accept one ore 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.

2.3.3. 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 E7XX_qPOS("123", 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 `E7XX_MOV("13", 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 `E7XX_MOV("112", 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 `E7XX_qPOS("112", pos[3])`, `pos[0]` and `pos[1]` will contain 2.0 as the position of '1'.

(See **E7XX_MOV**, p.24, **E7XX_qPOS**, p.25, or **E7XX_SEP**, p. 26)

2.4. Types Used in PI Software

2.4.1. 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
```

2.4.2. 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
```

2.4.3. 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`".

3. Controller Setup

3.1. System Parameter Settings

A wide range of system parameters, e.g. the stage parameters, are stored in the EPROM of the controller. When the stage is equipped with an ID-chip (is located in the stage connector) and connected to the controller for the first time, the stage parameters from the ID-chip will be written to the EPROM on controller power-on.

E7XX_qHPA (p. 24) gives a list of valid parameter numbers. Call **E7XX_SPA**() (p. 34) to modify parameters temporarily (to save them to EPROM use **E7XX_WPA**, p. 37), or **E7XX_SEP** (p. 33) to change the EPROM values. See "System Parameter Overview" on p. 47 for more information regarding the controller parameters and their handling.

The parameters in the ID-chip can not be overwritten. For stages with ID-chip the option "Read ID-Chip always" (parameter ID 0x0f000000) is disabled by default to make optimized parameter settings in the EPROM available in the future. See the User Manual of the controller for details.

3.2. Configuration of Axis

The following example shows how to connect to an E-761 (without the call printf()). In the example, **E7XX_qSAI**() (p. 26) is called to get the configured axes. For the configured axes no further initialization is required.

```
char axes[10];
int ID;

// connect to the E-761 over PCI (boardnumber 1)
ID = E7XX_ConnectPciBoard (1);
if (ID<0)
    return FALSE;

if (!E7XX_qSAI(ID, axes, 9))
    return FALSE;

// the output should be "12" - if axes 1 and 2 were configured
printf("qSAI() returned \"%s\"", axes);
```

Sometimes it might be necessary to change the axis configuration, e.g. when you disconnect stages from the controller or want to use axes which were not yet configured.

With **E7XX_qCST**() (p. 22) you can obtain a full list of the available axes on the controller and their current configuration—non-configured axes will show "NOSTAGE" as stage name, configured axes will show the name "ID-STAGE". In contrast to **E7XX_qCST**(), the function call **E7XX_qSAI**() (p. 26) returns only the axis identifiers of configured axes.

The following example shows how to change the axis configuration for an E-761 used with single-axis stages.

```
char stages[1024];
char axes[10];
int ID;

// connect to the E-761 over PCI (boardnumber 1)
ID = E7XX_ConnectPciBoard (1);
if (ID<0)
    return FALSE;

if (!E7XX_qCST(ID, "123", stages, 1023))
    return FALSE;

// If all axes are configured,
// the output should be "1=ID-STAGE \n2=ID-STAGE \n3=ID-STAGE\n"
printf("qCST() returned \"%s\"", stages);
```

```
if (!E7XX_qSAI(ID, axes, 9))
    return FALSE;

// the output should be "123" - axes 1, 2 and 3 are configured
printf("qSAI() returned \"%s\"", axes);

// Now only axes 1 and 2 are connected to the controller, so we have to set
// axes 3 to NOSTAGE.
sprintf(stages, "NOSTAGE");
if (!E7XX_CST(ID, "3", stages))
    return FALSE;

if (!E7XX_qSAI(ID, axes, 9))
    return FALSE;

// The output should be "12" - the new configured axes (axis 3 is now non-configured).
printf("qSAI() returned \"%s\"", axes);

if (!E7XX_qCST(ID, "123", stages, 1023))
    return FALSE;

// The output should be "1=ID-STAGE \n2=ID-STAGE \n3=NOSTAGE\n"
printf("qCST() returned \"%s\"", stages);
```

4. Communication Initialization

4.1. Functions

- int **E7XX_ConnectPciBoard**(long *iBoardNumber*)
- int **E7XX_ConnectPciBoardAndReBoot**(long *iBoardNumber*)
- BOOL **E7XX_IsConnected** (int *ID*)
- void **E7XX_CloseConnection** (int *ID*)
- int **E7XX_GetError** (int *ID*)
- BOOL **E7XX_TranslateError** (int *iErrorNumber*, char* *szErrorMessage*, int *iBufferSize*)
- BOOL **E7XX_SetErrorCheck** (int *ID*, BOOL *bErrorCheck*)

4.2. Detailed Description

To use the DLL and communicate with an E-7xx controller, the user must initialize the DLL with one of the "open" functions **E7XX_ConnectPciBoard**() or **E7XX_ConnectPciBoardAndReBoot**(). To allow the handling of multiple controllers, the user will be returned a non-negative "ID" when he calls one of these functions. 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 first argument. **E7XX_CloseConnection**()will close the connection to the specified controller and free its system resources.

4.3. Function Documentation

void **E7XX_CloseConnection** (int *ID*)

Close connection to E-7xx controller associated with *ID*. *ID* will not be valid after this call.

Valid for:

E-710, E-761

Arguments:

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

int **E7XX_ConnectPciBoard** (long *iBoardNumber*)

Open a PCI connection to an E-7xx PCI board. All future calls to control this E-7xx board need the ID returned by this call.

Valid for:

E-761

Arguments:

iBoardNumber number of board

Returns:

ID of new object, **-1** if interface could not be opened or no E-7xx is responding.

int **E7XX_ConnectPciBoardAndReBoot** (long *iBoardNumber*)

Open a PCI connection to an E-7xx PCI board and reboot it. All future calls to control this E-7xx board need the ID returned by this call.

Valid for:

E-761

Arguments:

iBoardNumber number of board

Returns:

ID of new object, **-1** if interface could not be opened or no E-7xx is responding.

int **E7XX_GetError** (int *ID*)

Get error status of the DLL and, if clear, that of the E-7xx. If the library shows an error condition, its code is returned, if not, the controller error code is checked using **E7XX_qERR**() (p.52) and retuned. After this call the DLL internal error state will be cleared; the controller error state will be cleared if it was queried.

Valid for:

E-710, E-761

Returns:

error ID, see **Error codes** (p.**52**) for the meaning of the codes.

BOOL **E7XX_IsConnected** (int *ID*)

Check if there is an E-7xx controller with an ID of *ID*.

Valid for:

E-710, E-761

Returns:

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

BOOL **E7XX_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 E-7xx (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 **E7XX_GetError**() (p.12) 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.

Valid for:

E-710, E-761

Arguments:

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

Returns:

the old state, before this call

BOOL **E7XX_TranslateError** (int *iErrorNumber*, char* *szErrorMessage*, int *iBufferSize*)

Translate error number to error message.

Valid for:

E-710, E-761

Arguments:

iErrorNumber number of error, as returned from **E7XX_GetError**()(p.12).
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

5. Fast Data Exchange

- BOOL **E761_SetDirectTarget** (int ID, const PI_E761_DIRECT_TARGET_WRITE piDirectTargetData)
- BOOL **E761_GetDirectPosition** (int ID, PI_E761_DIRECT_POSITION_READ *ppiDirectPositionData)

You can set control values directly in the E-761 RAM using the function **E761_SetDirectTarget**. To read positions directly from RAM, you can use the **E761_GetDirectPosition** library function.

NOTES

The E-761 board writes and reads data every servo loop, i.e. with up to 25 kHz. With every call of **E761_SetDirectTarget** or **E761_GetDirectPosition**, one value is written/read. Make sure that your system (hardware, operating system, configuration) is fast enough to attain the data rate provided by the E-761 board.

BOOL **E761_SetDirectTarget** (int *ID*, const PI_E761_DIRECT_TARGET_WRITE *piDirectTargetData*)

Set control values directly in the target register of the E-761 RAM. The E-761 board reads from the target register every servo loop, i.e. with up to 25 kHz.

Arguments:

ID ID of controller

piDirectTargetData the data which is to be written to the target register of the E-761 RAM:

fTargetAxis1: control value for axis 1
fTargetAxis2: control value for axis 2
fTargetAxis3: control value for axis 3
iEnableAxisFlags: enable/disable axis control via E761_SetDirectTarget as follows (see also table below):
EN_FAST_PCI_CH0 affects axis 1
EN_FAST_PCI_CH1 affects axis 2
EN_FAST_PCI_CH2 affects axis 3

31	30	29	28	27	26	25	24
r	EN_FAST_PCI_CH2	EN_FAST_PCI_CH1	EN_FAST_PCI_CH0	r	r	r	r
23	22	21	20	19	18	17	16
r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8
r	r	r	r	r	r	r	r
7	6	5	4	3	2	1	0
r	r	r	r	r	r	r	r

r = reserved; set to 0

The axis control value is defined as follows:

- When EN_FAST_PCI_CHn is 1 (enabled), the axis control value is taken from E-761 RAM, and any control values given by move commands or by the wave generator are overwritten. If the analog input is enabled for "direct" control of an axis (parameter ID 0x06000902 set with E7XX_SPA), its value is added to the current control value resulting from E761_SetDirectTarget.
- When EN_FAST_PCI_CHn is 0 (disabled), the axis control value is given by move commands, wave generator and/or analog input.

See "Control Value Generation" in the E-761 User Manual for more information.

Returns:

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

BOOL **E761_GetDirectPosition** (int *ID*, PI_E761_DIRECT_POSITION_READ **ppiDirectPositionData*)

Read positions directly from the position register of the E-761 RAM. The E-761 board writes to the position register every servo loop, i.e. with up to 25 kHz.

Arguments:

ID ID of controller

****ppiDirectPositionData*** is the data which is read from the position register of the E-761 RAM

fPositionAxis1: returns position of axis 1
fPositionAxis2: returns position of axis 2
fPositionAxis3: returns position of axis 3
fAnalogInput: returns voltage value of the analog input. Note that the voltage value is affected by the gain and offset settings made for that input channel (parameter ID 0x04000001 for the gain, parameter ID 0x04000101 for the offset).
iCounter: returns the counter value of the servo loop, will be increased with each servo loop (range is 0 to 255, if 255 is reached, counting restarts with 0). This value can be used to check whether new data is present or some data is lost.
iFlags: returns status flags as shown in the table below:

31	30	29	28	27	26	25	24
ERR	EN_FAST_ PCI_CH2	EN_FAST_ PCI_CH1	EN_FAST_ PCI_CH0	BSY	TRG	HVA3	HVA2
23	22	21	20	19	18	17	16
HVA1	HVA0	SEN2	SEN1	SEN0	ONT2	ONT1	ONT0
15	14	13	12	11	10	9	8
OVF2	OVF1	OVF0	WGO3	WGO2	WGO1	WGO0	r
7	6	5	4	3	2	1	0
r							

bit 31: ERR, can only be cleared by ERR? command
bits 30 to 28: enable flag for axis control, set by E761_SetDirectTarget
bit 27: BSY, system busy (command processing), can also be queried with the #7
bit 26: TRG, analog input state as trigger for the wave generator (if the voltage on the analog input is < 0.8 V, the signal is interpreted as LOW; if the voltage is ≥ 2.4 V, the signal is interpreted as HIGH)
bits 25 to 22: HVAn: high voltage amplifier is enabled to output
bits 21 to 19: SENn: sensor channel n enabled
bits 18 to 16: ONTn: on target, can also be queried with the ONT? command
bits 15 to 13: OVFn: overflow, can also be queried with the OVF? command
bits 12 to 9: WGOn: wave generator running, can also be queried with #9 or E7XX_IsGeneratorRunning
bit 8: r = reserved
bits 7 to 0: r = reserved

Returns:

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

6. Functions for GCS Commands

6.1. Motion and Controller Configuration

6.1.1. Function Overview

- BOOL **E7XX_ATZ** (int *ID*, const char* *szAxes*, const double* *pdLowVoltageArray*, const BOOL* *pfUseDefaultArray*)
- BOOL **E7XX_AVG** (int *ID*, int *iAverageTime*)
- BOOL **E7XX_CCL** (int *ID*, int *iCommandLevel*, const char**szPassWord*)
- BOOL **E7XX_CCT** (int *ID*, int *iCommandType*)
- BOOL **E7XX_CST** (int *ID*, const char* *szAxes*, const char* *szNames*)
- BOOL **E7XX_DFH** (int *ID*, const char* *szAxes*)
- BOOL **E7XX_GcsCommandset** (int *ID*, const char* *szCommand*)
- BOOL **E7XX_GcsGetAnswer** (int *ID*, char* *szAnswer*, int *iBufferSize*)
- BOOL **E7XX_GcsGetAnswerSize** (int *ID*, int* *iAnswerSize*)
- BOOL **E7XX_GOH** (int *ID*, const char* *szAxes*)
- BOOL **E7XX_HLT** (int *ID*, const char* *szAxes*)
- BOOL **E7XX_IMP** (int *ID*, char *cAxis*, double *dImpulseSize*)
- BOOL **E7XX_IMP_PulseWidth** (int *ID*, char *cAxis*, double *dImpulseSize*, int *iPulseWidth*)
- BOOL **E7XX_IsMoving** (const int *ID*, char *const *szAxes*, BOOL * *pbValueArray*)
- BOOL **E7XX_MOV** (int *ID*, const char* *szAxes*, const double* *pdValueArray*)
- BOOL **E7XX_MVR** (int *ID*, const char* *szAxes*, const double* *pdValueArray*)
- BOOL **E7XX_NLM** (int *ID*, const char* *szAxes*, const double* *pdValueArray*)
- BOOL **E7XX_PLM** (int *ID*, const char* *szAxes*, const double* *pdValueArray*)
- BOOL **E7XX_qAVG** (int *ID*, int* *piAverageTime*)
- BOOL **E7XX_qCCL** (int *ID*, int* *piComandLevel*)
- BOOL **E7XX_qCST** (int *ID*, const char* *szAxes*, char* *szNames*, int *iBufferSize*)
- BOOL **E7XX_qCSV** (int *ID*, double *pdCommandSyntaxVersion*
- BOOL **E7XX_qDFH** (int *ID*, const char* *szAxes*, double* *pdValueArray*)
- BOOL **E7XX_qDIO** (int *ID*, const char**szChannels*, BOOL* *pbValueArray*)
- BOOL **E7XX_qDRR_SYNC** (int *ID*, int *piRecordChannelId*, int *iOffsetOfFirstPointInRecordTable*, int *iNumberOfValues*, double* *pdValueArray*)
- BOOL **E7XX_qERR** (int *ID*, int* *piError*)
- BOOL **E7XX_qHLP** (int *ID*, char* szBuffer, int *iBufferSize*)
- BOOL **E7XX_qHPA** (int *ID*, char* *szBuffer*, int *iBufferSize*)
- BOOL **E7XX_qIDN** (int *ID*, char* *szBuffer*, int *iBufferSize*)
- BOOL **E7XX_qIMP** (int *ID*, char *cAxis*, int *iOffsetOfFirstPointInRecordTable*, int *iNumberOfValues*, double* *pdValueArray*)
- BOOL **E7XX_qMOV** (int *ID*, const char* *szAxes*, double* *pdValueArray*)
- BOOL **E7XX_qNLM** (int *ID*, const char* *szAxes*, double* *pdValueArray*)
- BOOL **E7XX_qONT** (int *ID*, const char* *szAxes*, int* *piValueArray*)
- BOOL **E7XX_qOVF** (int *ID*, const char* *szAxes*, int* *piValueArray*)
- BOOL **E7XX_qPLM** (int *ID*, const char* *szAxes*, double* *pdValueArray*)
- BOOL **E7XX_qPOS** (int *ID*, const char* *szAxes*, double* *pdValueArray*)
- BOOL **E7XX_ qRTR** (int *ID*, long *iReportTableRate*)
- BOOL **E7XX_qSAI** (int *ID*, char* *szAxes*, int *iBufferSize*)
- BOOL **E7XX_qSAI _ALL** (int *ID*, char* *szAxes*, int *iBufferSize*)
- BOOL **E7XX_qSEP** (int *ID*, const char* *szAxes*, const long* *iParameterArray*, double* *pdValueArray*, char* *szStrings*, long *iMaxStringSize*)
- BOOL **E7XX_qSPA** (int *ID*, const char* *szAxes*, const int* *iParameterArray*, double* *pdValueArray*, char* *szStrings*, int *iMaxStringSize*)
- BOOL **E7XX_ qSSN** (int *ID*, char* *szSerialNumber*, int *iBufferSize*)
- BOOL **E7XX_qSTE** (int *ID*, char *cAxis*, int *iOffsetOfFirstPointInRecordTable*, int *iNumberOfValues*, double* *pdValueArray*)
- BOOL **E7XX_qSVA** (int *ID*, const char* *szAxes*, double* *pdValueArray*)
- BOOL **E7XX_qSVO** (int *ID*, const char* *szAxes*, BOOL* *pbValueArray*)
- BOOL **E7XX_qTAD** (int *ID*, const char**szSensorChannels*, int* *piValueArray*)

- BOOL **E7XX_ qTAV** (int *ID*, const char**szAnalogInputChannels*, int* *piValueArray*)
- BOOL **E7XX_ qTIO** (int *ID*, int* *pInputNr*, int* *pOutputNr*)
- BOOL **E7XX_qTMN** (int *ID*, const char* *szAxes*, double* *pdValueArray*)
- BOOL **E7XX_qTMX** (int *ID*, const char* *szAxes*, double* *pdValueArray*)
- BOOL **E7XX_qTNR** (int *ID*, int* *piRecordChannels*)
- BOOL **E7XX_qTNS** (int *ID*, const char**szSensorChannels*, int* *piValueArray*)
- BOOL **E7XX_qTPC** (int *ID*, int* *piPiezoChannels*)
- BOOL **E7XX_qTSC** (int *ID*, int* *piSensorChannels*)
- BOOL **E7XX_qTSP** (int *ID*, const char**szSensorChannels*, int* *piValueArray*)
- BOOL **E7XX_qTVI** (int *ID*, char* szAxes, int *iBufferSize*)
- BOOL **E7XX_qVCO** (int *ID*, char* *szAxes*, BOOL* *pbValueArray*)
- BOOL **E7XX_qVEL** (int *ID*, const char* *szAxes*, double* *pdValueArray*)
- BOOL **E7XX_qVER** (int *ID*, char* *szVersion*, int *iBufferSize*)
- BOOL **E7XX_qVMA** (int *ID*, const char* *szPiezoChannels*, double* *pdValueArray*)
- BOOL **E7XX_qVMI** (int *ID*, const char* *szPiezoChannels*, double* *pdValueArray*)
- BOOL **E7XX_qVOL** (int *ID*, const char* *szPiezoChannels*, double* *pdValueArray*)
- BOOL **E7XX_qVST** (int *ID*, char* *szValideStages*, int *iBufferSize*)
- BOOL **E7XX_RBT** (const int *ID*)
- BOOL **E7XX_RPA** (int *ID*, const char* *szAxes*, const long* *iParameterArray*)
- BOOL **E7XX_RTR** (int *ID*, long *iReportTableRate*)
- BOOL **E7XX_SAI** (int *ID*, const char* *szOldAxes*, const char* *szNewAxes*)
- BOOL **E7XX_SEP** (int I*D*, const char**szPassword*, const char* *szAxes*, const long* *iParameterArray*, const double* *pdValueArray*, const char* *szStrings*)
- BOOL **E7XX_SPA** (int *ID,* const char* *szAxes*, const int* *iParameterArray*, const double* *pdValueArray*, const char* *szStrings*)
- BOOL **E7XX_STE** (int *ID*, char *cAxis*, double *dStepSize*)
- BOOL **E7XX_STP** (int *ID*, const char* *szAxes*)
- BOOL **E7XX_SVA** (int *ID*, const char* *szAxes*, const double* *pdValueArray*)
- BOOL **E7XX_SVO** (int *ID*, const char* *szAxes*, const BOOL* *pbValueArray*)
- BOOL **E7XX_SVR** (int *ID*, const char* *szAxes*, const double* *pdValueArray*)
- BOOL **E7XX_VCO** (int *ID*, char* *szAxes*, const BOOL* *pbValueArray*)
- BOOL **E7XX_VEL** (int *ID*, const char* *szAxes*, const double* *pdValueArray*)
- BOOL **E7XX_VMA** (int *ID*, const char* *szPiezoChannels*, const double* *pdValueArray*)
- BOOL **E7XX_VMI** (int *ID*, const char* *szPiezoChannels*, const double* *pdValueArray*)
- BOOL **E7XX_VOL** (int *ID*, const char* *szPiezoChannels*, const double* *pdValueArray*)
- BOOL **E7XX_WPA** (int *ID*, const char**szPassword*, const char* *szAxes*, const long* *iParameterArray*)

6.1.2. Function Documentation

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

BOOL **E7XX_ATZ** (int *ID*, const char* *szAxes*, const double* *pdLowVoltageArray*, const BOOL* *pbUseDefaultArray*)

Corresponding **command:** ATZ

Performs an automatic zero-point calibration for szAxes. Each linear axis listed will be autozeroed whether autozero is enabled for the axis or not. Rotation axes will not be affected. This procedure lasts several seconds. The controller will be "busy" during AutoZero, so most other commands will cause a **PI_CONTROLLER_BUSY** error. ATZ works independent of servo mode. Just after execution the current position is 0.

Be aware that the result of the AutoZero procedure (new offset value) is automatically written to non-volatile memory (EPROM). For stages with ID-chip the option "Read ID-Chip always" must be disabled by default to make the AutoZero result (new offset value) available in the future. See E-761 User Manual for details.

Arguments:

ID ID of controller

szAxes string with axes

pdLowVoltageArray Array with low voltages for the corresponding axes.

pbUseDefaultArray If TRUE the value in *pdLowVoltageArray* for the axis is ignored and the value stored in the controller is used.

Returns:

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

BOOL **E7XX_AVG** (int *ID*, int *iAverageTime*)

Corresponding command: AVG

Set the number of ADC samples used to calculate an average (oversampling factor).

Notes:

A higher value increases the values for "Sensor sampling time" (ID 0x0e000100) and "Servo update time" (ID 0x0e000200) while keeping the original ADC sampling frequency constant.

The values influenced by this command are saved in RAM only. To save the currently valid parameters to flash ROM, where they become the power-on defaults, you must use **E7XX_WPA** (p. 37). Parameter changes not saved with **E7XX_WPA** will be lost when the controller is powered off.

Arguments:

ID ID of controller

iAverageTime number of samples used for average (oversampling factor), must be one of 4, 8, 16, 32, 64, 128

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **E7XX_CCL** (int *ID*, int *iCommandLevel*, const char**szPassWord*)

Corresponding **command:** CCL

If *Password* is correct, this function sets the *CommandLevel* of the controller and determines thus the availability of commands and the write access to the system parameters. Use **E7XX_qHLP** to determine which commands are available in the current command level.

Arguments:

ID ID of controller

iCommandLevel can be

0 (only commands needed for normal operation are available)

1 (all commands from command level 0 plus special commands for advanced users are available)

szPassword password for CCL 1 is “ADVANCED”, for CCL 0 no password is required

Returns:

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

BOOL **E7XX_CCT** (int *ID*, int *iCommandType*)

Corresponding **command:** CCT

Change command type by switching between binary and ASCII protocol. The command type is changed immediately after the function was called.

Note: After power up the controller communicates via ASCII protocol.

Arguments:

ID ID of controller

iCommandType is the command type to use, 1 = binary, 2 = string

Returns:

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

BOOL **E7XX_CST** (int *ID*, const char* *szAxes*, const char* *szNames*)

Corresponding **command:** CST

Set the type names of the stages associated with *szAxes*. The individual names are separated by '\n' ("line-feed"), for example "ID-STAGE\n NOSTAGE". For a list of existing stage names call **E7XX_qVST**() (p. 32). E7XX_CST must be called before you can address the connected stages. See “Controller Setup” (p. 9) for an example how to set up the E-7XX library.

Notes:

When you add or replace stages and configure the axes with CST, the ID-chips of the connected stages are not read by the controller yet. To read the ID-chip data, the controller must be rebooted.

The configuration made with CST is automatically saved to the non-volatile memory (EEPROM).

Arguments:

ID ID of controller

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

szNames the *names* of the stages separated by '\n' ("line-feed"); "ID-STAGE for configured axes (a stage should be connected), "NOSTAGE" for non-configured axes (no stage should be connected)

Returns:

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

BOOL **E7XX_DFH** (int *ID*, const char* *szAxes*)

Corresponding **command:** DFH

Makes current positions of *szAxes* the new home positions

Arguments:

ID ID of controller

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

Returns:

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

BOOL **E7XX_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.

Arguments:

ID ID of controller

szCommand the GCS command as string.

Returns:

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

BOOL **E7XX_GcsGetAnswer** (int *ID*, char* *szAnswer*, int *iBufferSize*)

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

Arguments:

ID ID of controller

szAnswer the buffer to receive the answer.

iBufferSize the size of *szAnswer*.

Returns:

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

BOOL **E7XX_GcsGetAnswerSize** (int *ID*, int* *iAnswerSize*)

Gets the size of an answer of a GCS command.

Arguments:

ID ID of controller

iAnswerSize pointer to integer to receive the size of the oldest answer waiting in the library.

Returns:

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

BOOL **E7XX_GOH** (int *ID*, const char* *szAxes*)

Corresponding **command:** `GOH`

Move all axes in *szAxes* to their home positions.

Arguments:

ID ID of controller

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

Returns:

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

BOOL **E7XX_HLT** (int *ID*, const char* *szAxes*)

Corresponding **command:** `HLT`

Halt the motion of given axes smoothly. Only non-complex motion (e.g. E7XX_MOV, E7XX_GOH, E7XX_SVR, E7XX_STE) can be interrupted with E7XX_HLT. Error code 10 is set. After the stage was stopped, the target position is set to the current position.

E7XX_HLT does not take effect to analog input which is used for "direct" axis control (see E-761 User Manual for more information). To disable "direct" control for an axis, the value of the corresponding "Aux-Input to target factor" parameter (ID 0x06000902) must be set to 0 with E7XX_SPA.

Arguments:

ID ID of controller

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

Returns:

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

BOOL **E7XX_IMP** (int *ID*, char *cAxis*, double *dImpulseSize*)

Corresponding command: `IMP`

Record impulse response for one axis. The controller will move the given axis relative to the current position and record 8192 position values from start.

The number of servo cycles used for data recording depends on the setting made with **E7XX_RTR** (p. 32). Call **E7XX_qDRR_SYNC**() (p. 23) or **E7XX_qIMP**() (p.24) to read the recorded data.

Arguments:

ID ID of controller

cAxis axis for which the impulse response will be recorded

dImpulseSize pulse height.

Returns:

TRUE if no error **FALSE** otherwise

BOOL **E7XX_IMP_PulseWidth** (int *ID*, char *cAxis*, double *dImpulseSize*, int *iPulseWidth*)

Corresponding command: `IMP`

Record impulse response for one axis. The controller will move the given axis relative to the current position and record 8192 position values from start.

The number of servo cycles used for data recording depends on the setting made with **E7XX_RTR** (p. 32). Call **E7XX_qDRR_SYNC**() (p. 23) or **E7XX_qIMP**() (p.24) to read the recorded data.

Arguments:

ID ID of controller
cAxis axis for which the impulse response will be recorded
dImpulseSize pulse height.
iPulseWidth the pulse width in cycle times.

Returns:

TRUE if no error **FALSE** otherwise

BOOL **E7XX_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 *pbValarray[0]* will contain a generalized state: TRUE if at least one axis is moving, FALSE if no axis is moving.

Arguments:

ID ID of controller
szAxes string with axes, if "" or **NULL** all axes are affected.
pbValueArray status of the axes

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **E7XX_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. Servo must be enabled for all commanded axes prior to using this command.

Arguments:

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

Returns:

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

BOOL **E7XX_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.

Arguments:

ID ID of controller
szAxes string with axes
pdValueArray amounts to be added (algebraically) to current target positions of the axes

Returns:

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

BOOL **E7XX_NLM** (int *ID*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: `NLM`

Set lower limits ("soft limit") for the positions of *szAxes*. Settings made with E7XX_NLM are only valid for closed-loop operation (servo ON).

Note:

This function saves the parameters in RAM only. To save the currently valid parameters to flash ROM, where they become the power-on defaults, you must use **E7XX_WPA** (p. 37) without any parameter. Changes not saved with **E7XX_WPA** will be lost when the controller is powered off.

Arguments:

ID ID of controller
szAxes string with axes
pdValueArray lower limits for position

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **E7XX_PLM** (int *ID*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: `PLM`

Set upper limits ("soft limit") for the positions of *szAxes*. Settings made with E7XX_PLM are only valid for closed-loop operation (servo ON).

Note:

This function saves the parameters in RAM only. To save the currently valid parameters to flash ROM, where they become the power-on defaults, you must use **E7XX_WPA** (p. 37) without any parameters. Changes not saved with **E7XX_WPA** will be lost when the controller is powered off.

Arguments:

ID ID of controller
szAxes string with axes
pdValueArray upper limits for position

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **E7XX_qAVG** (int *ID*, int* *piAverageTime*)

Corresponding command: `AVG?`

Get the number of samples used for average.

Arguments:

ID ID of controller
piAverageTime pointer to `int` for storing the number of samples used for average

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **E7XX_qCCL** (int *ID*, const char* *CommandLevel*)

Corresponding **command:** `CCL?`

Returns the current *CommandLevel*.

Arguments:

ID ID of controller
CommandLevel variable to receive the current command level. See **E7XX_CCL** for possible values.

Returns:

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

BOOL **E7XX_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. For example "A=ID-STAGE \nB=NOSTAGE2\n".

Arguments:

ID ID of controller
szAxes identifiers of the axes, if "" or **NULL** all axes are queried
szNames buffer to receive the string read in from controller, lines are separated by '\n' ("line-feed")
iBufferSize size of *szNames*, must be given to avoid a buffer overflow.

Returns:

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

BOOL **E7XX_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. 7)

BOOL **E7XX_qDFH** (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding **command:** DFH?

Get the distance between the home position and the hardware origin for *szAxes*.

Arguments:

ID ID of controller
szAxes string with axes, if "" or **NULL** all axes are queried.
pdValueArray array to receive the home position displacements of the axes

Returns:

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

BOOL **E7XX_qDIO** (int *ID*, const char**szChannels*, BOOL* *pbValueArray*)

Corresponding **command:** DIO?

Returns the states of the specified Digital Input channels.

Notes:

Use **E7XX_qTIO** (p. 29) to get the number of installed digital I/O channels.

Note that the E-761 has no genuine digital input lines, but the analog input is internally interpreted as digital input for triggering tasks (see E-761 User Manual), and its signal state can be queried by this command. If the voltage on the analog input is < 0.8 V, the signal is interpreted as LOW; if the voltage is ≥ 2.4 V, the signal is interpreted as HIGH.

Arguments:

ID ID of controller
szChannels string with digital input channels
pbValueArray array to be filled with current status of the digital input channels (0 = digital input is LOW/OFF, 1 = digital input is HIGH/ON)

Returns:

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

BOOL **E7XX_qDRR_SYNC** (int *ID*, int *iRecordChannelId*, int *iOffsetOfFirstPointInRecordTable*, int *iNumberOfValues*, double* *pdValueArray*)

Corresponding **command:** DRR?

Returns N recorded data points. N must be less than or equal to Nmax.

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.

Recording takes place for all recorder tables in the following cases:

- The wave generator is running for an arbitrary axis; recording starts either automatically when the wave generator is started with **E7XX_WGO** (p. 45), or can be started "manually" with **E7XX_WGR** (p. 46).
- An impulse is started with **E7XX_IMP** (p. 19) or **E7XX_IMP_PulseWidth** (p. 19) or a step is started with **E7XX_STE** (p. 34).

The assignment of axis and data sources to the recorder tables is as follows:

table 1: axis 1 actual position
table 2: axis 2 actual position
table 3: axis 3 actual position
table 4: analog input voltage (same value as read with **E7XX_qTAV**, i.e. contains gain and offset for the analog input, see p. 28)

Immediately after recording is started, the record table which is currently filled may still contain some data from the previous run. Within a period of

time [ms] = oversampling factor * wave length [no. of points] * 10-2

you should therefore not query the data to avoid display errors (the oversampling factor can be queried using **E7XX_qAVG**, p. 21, the wave length depends on the wave definition).

The recorded data is influenced by the setting for the record table rate (**E7XX_RTR**, p. 32).

Arguments:

ID ID of controller
iRecordChannelId Id of the record table.
iOffsetOfFirstPointInRecordTable The start point in the specified record table
iNumberOfValues The number of values to read.
pdValueArray array to receive the values

Returns:

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

BOOL **E7XX_qERR** (int *ID*, int* *piError*)

Corresponding **command:** ERR?

Get the error state of the controller. Because the library may have queried (and cleared) controller error conditions on its own, it is safer to call **E7XX_GetError**()(p.12) which will first check the internal error state of the library. For a list of possible error codes see p. 52.

Arguments:

ID ID of controller
piError integer to receive error code of the controller

Returns:

TRUE if query successful, **FALSE** otherwise

BOOL **E7XX_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 *buffer*, must be given to avoid buffer overflow.

Returns:

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

BOOL **E7XX_qHPA** (int *ID*, char* *szBuffer*, int *iBufferSize*)

Corresponding **command:** `HPA?`

Returns a help string containing information about valid parameter numbers. Valid parameters depend on the current command level (ask with **E7XX_qCCL**).

See “System Parameter Overview,” beginning on p. 47, for a list of valid parameter numbers for command level 0, and for details regarding the controller parameters and their handling.

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 *buffer*, must be given to avoid buffer overflow.

Returns:

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

BOOL **E7XX_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 *buffer*, must be given to avoid a buffer overflow.

Returns:

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

BOOL **E7XX_qIMP** (int *ID*, char *cAxis*, int *iOffsetOfFirstPointInRecordTable*, int *iNumberOfValues*, double* *pdValueArray*)

Corresponding **command:** `IMP?`

Get the recorded positions of an impulse response. **E7XX_IMP**() (p.19) or **E7XX_IMP_PulseWidth**() (p. 19) must have been called to run and record the impulse response measurement.

Arguments:

ID ID of controller
cAxis axis for which the recorded impulse response is to be read
iOffsetOfFirstPointInRecordTable index of first value to be read. (The first stored value has index 0.)
iNumberOfValues number of values to be read. At most 8192 positions are stored.
pdValueArray Array to store the position values. Caller is responsible for providing enough space for *nrValues* doubles

Returns:

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

Errors:

PI_INVALID_ARGUMENT the combination of *iOffset* and *nrValues* includes values out of range

BOOL **E7XX_qMOV** (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding **command:** `MOV?`

Read the commanded target positions for *szAxes*.

Arguments:

ID ID of controller
szAxes string with axes, if "" or **NULL** all axes are queried.
pdValueArray array to be filled with target positions of the axes

Returns:

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

BOOL **E7XX_qNLM** (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: NLM?

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

Arguments:

ID ID of controller
szAxes string with axes
pdValueArray array to be filled with lower limits for position of the axes.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **E7XX_qONT** (int *ID*, const char* *szAxes*, BOOL* *piValueArray*)

Corresponding command: ONT?

Check if *szAxes* have reached target position. The axis is on target when the current position reaches a certain settle window around the target position. The size of the settle window for an axis depends on the "Tolerance" parameter (parameter ID 0x07000900).

Arguments:

ID ID of controller
szAxes string with axes
piValueArray array to be filled with current on-target status of the axes

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **E7XX_qOVF** (int *ID*, const char* *szAxes*, BOOL* *piValueArray*)

Corresponding command: OVF?

Checks overflow status of *szAxes*. Overflow means that the control variables are out of range (can only happen if controller is in closed-loop mode).

Arguments:

ID ID of controller
szAxes string with axes
piValueArray array to be filled with current overflow status of the axes

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **E7XX_qPLM** (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: PLM?

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

Arguments:

ID ID of controller
szAxes string with axes
pdValueArray array to be filled with upper limits for position of the axes.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **E7XX_qPOS** (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: POS?

Get the current positions of *szAxes*.

The sensor sampling time is influenced by **E7XX_AVG** (p. 17).

Arguments:

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

pdValueArray array to receive the current positions of the axes

Returns:

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

BOOL **E7XX_qRTR** (int *ID*, long *iReportTableRate*)

Corresponding command: RTR?

Reads the record table rate, i.e. the number of servo-loop cycles to be used in data recording operations.

Arguments:

ID ID of controller

iReportTableRate variable to filled with the table rate

Returns:

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

BOOL **E7XX_qSAI** (int *ID*, char* *szAxes*, int *iBufferSize*)

Corresponding command: SAI?

Get the single-character identifiers for all configured axes. Each character in the returned string is an axis identifier for one logical axis.

Do not mistake the axis identifiers set with E7XX_SAI with the Axis name parameter (ID 0x07000600) which is only used in the graphical user interface of NanoCapture™.

Arguments:

ID ID of controller

szAxes buffer to receive the string read in

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

Returns:

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

BOOL **E7XX_qSAI _ALL** (int *ID*, char* *szAxes*, int *iBufferSize*)

Corresponding command: SAI?

Get the single-character identifiers for all axes (configured and unconfigured axes). Each character in the returned string is an axis identifier for one logical axis.

Do not mistake the axis identifiers set with E7XX_SAI with the Axis name parameter (ID 0x07000600) which is only used in the graphical user interface of NanoCapture™.

Arguments:

ID ID of controller

szAxes buffer to receive the string read in

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

Returns:

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

BOOL **E7XX_qSEP** (int *ID*, const char* *szAxes*, const int* *piParameterArray*, double* *pdValueArray*, char* *szStrings*, int *iMaxStringSize*)

Corresponding command: SEP?

Query specified parameters for *szAxes* from EPROM. 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. 47, for a list of valid parameter numbers.

Arguments:

ID ID of controller

szAxes string with designator, one parameter is read for each designatorID in *szAxes*
for axis-related parameters: axis name;
for piezo- or sensor-related parameters: channel number;
otherwise a parameter-related code

piParameterArray parameter numbers

pdValueArray array to receive the values of the requested parameters
szStrings string to receive the with linefeed-separated parameter values (e.g. "X \nμm\n" are parameters 0x07000600 and 0x07000601); when not needed set to **NULL** (i.e. if numeric parameter values are queried)
iMaxStringSize size of *szStrings*, must be given to avoid a buffer overflow.

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

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

BOOL **E7XX_qSPA** (int *ID*, const char* *szAxes*, constz int* *piParameterArray*, double* *pdValArray*, char* *szStrings*, int *iMaxStringSize*)

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. 47, for a list of valid parameter numbers.

Arguments:
ID ID of controller
szAxes string with designator, one parameter is read for each designator in *szAxes*
for axis-related parameters: axis name;
for piezo- or sensor-related parameters: channel number;
otherwise a parameter-related code
piParameterArray parameter numbers
pdValArray array to be filled with the values of the requested parameters
szStrings string to receive the linefeed-separated parameter values (e.g. "X \nμm\n" are parameters 0x07000600 and 0x07000601); when not needed set to **NULL** (i.e. if numeric parameter values are queried)
iMaxStringSize size of *szStrings*, must be given to avoid a buffer overflow.

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

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

BOOL **E7XX_qSSN** (int *ID*, char* *szSerialNumber*, int *iBufferSize*)

Corresponding command: SSN?

Get serial number of the controller.

Arguments:
ID ID of controller
szSerialNumber buffer for storing the string read in
iBufferSize size of *buffer*, must be given to avoid buffer overflow.

Returns:
TRUE if successful, **FALSE** otherwise

BOOL **E7XX_qSTE** (int *ID*, const char *cAxis*, int *iOffset*, int *nrValues*, double* *pdValueArray*)

Corresponding command: STE?

Get the recorded positions of a step response. **E7XX_STE**() (p.27) must have been called before to start and record the step response measurement.

Arguments:
ID ID of controller
cAxis axis for which the step response values have been recorded
iOffset index of first value to be read (the first stored value has index 0)
nrValues number of values to read. At most 8192 positions are stored.

pdValueArray Array to receive the position values. Caller is responsible for providing enough space for *nrValues* doubles

Returns:

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

Errors:

PI_INVALID_ARGUMENT the combination of *iOffset* and *nrValues* specifies values out of range

BOOL **E7XX_qSVA** (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: SVA?

Read the commanded open-loop control values for ***szAxes***.

Arguments:

ID ID of controller

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

pdValueArray array to be filled with the open-loop control values for the axes

Returns:

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

BOOL **E7XX_qSVO** (int *ID*, const char* szAxes, BOOL* *pbValueArray*)

Corresponding command: SVO?

Get the servo-control mode for ***szAxes***

Arguments:

ID ID of controller

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

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

Returns:

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

BOOL **E7XX_qTAD** (int *ID*, const char**szSensorChannels*, int* *piValueArray*)

Corresponding command: TAD?

Returns AD value for the specified sensor number.

Arguments:

ID ID of controller

szSensorChannels string with sensors, if "" or **NULL** all sensors are queried.

piValueArray array to receive AD value (dimensionless)

Returns:

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

BOOL **E7XX_qTAV** (int *ID*, const char**szAnalogInputChannels*, int* *piValueArray*)

Corresponding command: TAV?

Returns voltage value for the specified analog input channel. Note that the voltage value is affected by the gain and offset settings made for that input channel (parameter ID 0x04000001 for the gain, parameter ID 0x04000101 for the offset).

Arguments:

ID ID of controller

szAnalogInputChannels string with channels, valid specification is "a4" since the E-761 has only one analog input line (which is internally handled as the 4th channel of the A/D converter). If "" or **NULL** all analog input channels are queried.

piValueArray array to receive voltage value (in volts)

Returns:

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

BOOL **E7XX_qTIO** (int *ID*, int* *pInputNr*, int* *pOutputNr*)

Corresponding command: TIO?

Returns the number of available digital I/O channels.

Note:

The E-761 has no genuine digital input and output lines, but the analog input is internally interpreted as digital input for triggering tasks (see E-761 User Manual), and its signal state can be queried with **E7XX_qDIO** (p. 22).

Arguments:

ID ID of controller
pInputNr variable to receive number of available digital input channels
pOutputNr variable to receive number of available digital output channels

Returns:

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

BOOL **E7XX_qTMN** (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: TMN?

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

Arguments:

ID ID of controller
szAxes string with axes, if "" or **NULL** all axes are queried.
pdValueArray array to receive low end of the travel range of the axes

Returns:

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

BOOL **E7XX_qTMX** (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: TMX?

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

Arguments:

ID ID of controller
szAxes string with axes, if "" or **NULL** all axes are queried
pdValueArray array to receive high end of travel range of the axes

Returns:

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

BOOL **E7XX_qTNR** (int *ID*, int* *piRecordChannels*)

Corresponding command: TNR?

Returns the number of recording tables.

Arguments:

ID ID of controller
piRecordChannels variable to receive number of recording tables

Returns:

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

BOOL **E7XX_qTNS** (int *ID*, const char**szSensorChannels*, double* *pdValueArray*)

Corresponding command: TNS?

Returns norminized sensor value for the specified sensor number.

Arguments:

ID ID of controller
szSensorChannels string with sensors, if "" or **NULL** all sensors are queried.
pdValueArray array to receive nom. sensor value (dimensionless)

Returns:

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

BOOL **E7XX_qTPC** (int *ID*, int* *piPiezoChannels*)

Corresponding command: TPC?

Returns the number of available piezo channels.

Arguments:

ID ID of controller

piPiezoChannels variable to receive number of available piezo channels

Returns:

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

BOOL **E7XX_qTSC** (int *ID*, int* *piSensorChannels*)

Corresponding command: TSC?

Returns the number of available sensor channels.

Arguments:

ID ID of controller

piSensorChannels variable to receive number of sensor channels

Returns:

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

BOOL **E7XX_qTSP** (int *ID*, const char**szSensorChannels*, double* *pdValueArray*)

Corresponding command: TSP?

Returns sensor position for the specified sensor number.

Arguments:

ID ID of controller

szSensorChannels string with sensors, if "" or **NULL** all sensors are queried.

pdValueArray array to receive sensor position (in µm or µrad)

Returns:

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

BOOL **E7XX_qTVI** (int *ID*, char* *szAxes*, int *iBufferSize*)

Corresponding command: TVI?

Get valid identifiers for axes. Each character in the returned string is a valid axis identifier that can be used to designate an axis in other commands.

Arguments:

ID ID of controller

szAxes buffer to receive the identifiers of the axes

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

Returns:

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

BOOL **E7XX_qVCO** (int *ID*, char* *szAxes*, BOOL* *pbValueArray*)

Corresponding command: VCO?

Get the velocity-control mode for *szAxes*. During wave generator output, there is no velocity control, i.e. any settings made with E7XX_VCO will be ignored.

Arguments:

ID ID of controller

szAxes string with axes

pbValueArray array to be filled with the velocity-control modes of the specified axes, **TRUE** for "on", **FALSE** for "off"

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **E7XX_qVEL** (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: VEL?

Get the velocity settings of *szAxes*.

Arguments:

ID ID of controller
szAxes string with axes, if "" or **NULL** all axes are queried.
pdValueArray array to be filled with the velocity settings of the axes

Returns:

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

BOOL **E7XX_qVER** (int *ID*, char* *szBuffer*, int *iBufferSize*)

Corresponding command: VER?

Reports device identity number and DSP firmware version.

Arguments:

ID ID of controller
szBuffer buffer for storing the string read in
iBufferSize size of *buffer*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **E7XX_qVMA** (int *ID*, const char* *szPiezoChannels*, double* *pdValueArray*)

Corresponding command: VMA?

Get upper limits for the PZT voltage for *szPiezoChannels*.

Arguments:

ID ID of controller
szPiezoChannels string with PZT channels
pdValueArray array to be filled with the upper limits for PZT voltage for the PZT channels

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **E7XX_qVMI** (int *ID*, const char* *szPiezoChannels*, double* *pdValueArray*)

Corresponding command: VMI?

Get lower limits for the PZT voltage for *szPiezoChannels*.

Arguments:

ID ID of controller
szPiezoChannels string with PZT channels
pdValueArray array to be filled with the lower limits for PZT voltage for the PZT channels

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **E7XX_qVOL** (int *ID*, const char* *szPiezoChannels*, double* *pdValueArray*)

Corresponding command: VOL?

Get current PZT voltages for *szPiezoChannels*

Arguments:

ID ID of controller
szPiezoChannels string with PZT channels, if "" or **NULL** all PZT channels are queried
pdValueArray array to be filled with the current voltages for the PZT channels

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

BOOL **E7XX_qVST** (int *ID*, char* *szBuffer*, int *iBufferSize*)

Corresponding command: VST?

List the stage names which can be used for the axis configuration with **E7XX_CST**.

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 *buffer*, must be given to avoid a buffer overflow.

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

BOOL **E7XX_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 **E7XX_RPA** (int *ID*, const char* *szAxes*, const long* *piParameterArray*)

Corresponding command: RPA

Copy specified parameters for *szAxes* from the EPROM 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 *iParameterArray*. See “System Parameter Overview,” beginning on p. 47, for a list of valid parameter numbers.

Arguments:
ID ID of controller
szAxes string with designators, one parameter is copied for each designator in *szAxes*
for axis-related parameters: axis identifier;
for piezo- or sensor-related parameters: channel number;
otherwise a parameter-related code
piParameterArray parameter numbers

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

BOOL **E7XX_RTR** (int *ID*, long *piReportTableRate*)

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
piReportTableRate is the record table rate to be used (unit: number of servo-loop cycles), must be larger than zero

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

BOOL **E7XX_SAI** (int *ID*, const char* *szOldAxes*, const char* *szNewAxes*)

Corresponding command: SAI

Assign new identifiers to axes (axes must have been configured with **E7XX_CST** before). *szOldAxes[index]* will be set to *szNewAxes[index]*. The characters in *szNewAxes* must not be in use for any other existing axes and must be one of the valid identifiers. All characters in *szNewAxes* will be converted to uppercase letters. To find out which characters are valid, call **E7XX_qTVI**() (p.30). If the same axis identifier occurs more than once in *szOldAxes,* only the **last** occurrence will be used to change the name.

The configuration made with E7XX_SAI is automatically saved to the non-volatile memory (EPROM). Do not mistake the axis identifiers set with E7XX_SAI with the Axis name parameter (ID 0x07000600) which is only used in the graphical user interface of NanoCapture™.

Arguments:

ID ID of controller
szOldAxes string with axes whose identifiers are to be changed (old identifiers)
szNewAxes new identifiers for the respective axes

Returns:

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

Errors:

PI_INVALID_AXIS_IDENTIFIER one or more characters not valid
PI_UNKNOWN_AXIS_IDENTIFIER if *szOldAxes* contains unknown axis
PI_AXIS_ALREADY_EXISTS one or more characters in *szNewAxes* is already in use as axis ID
PI_INVALID_ARGUMENT if *szOldAxes* and *szNewAxes* have different lengths or if a character in *szNewAxes* is used for more than one old axis

BOOL **E7XX_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 EPROM. 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. 47, for a list valid parameter numbers.

Notes:

If the same designator has the same parameter number more than once, only the **last** value will be set. For example `E7XX_SEP(id, "100", "111", {0x07000300, 0x07000300, 0x07000301}, {3e-2, 2e-2, 2e-4})` will set the P-term of '1' to 2e-2 and the I-term to 2e-4.
E7XX_SEP only writes to non-volatile memory. After parameters were set with **E7XX_SEP**, use **E7XX_RPA** (p. 32) to activate them (write them to volatile memory), or they become active after next reboot.

Arguments:

ID ID of controller
szPassword There is a password required to set parameters in the EPROM . This password is "100"
szAxes string with designators, one parameter is set for each designator in *szAxes*
for axis-related parameters: axis identifier;
for piezo- or sensor-related parameters: channel number;
otherwise a parameter-related code
piParameterArray Parameter numbers
pdValueArray array with the values for the respective parameters
szStrings string with linefeed-separated parameter values (e.g. "X \nμm\n" are parameters 0x07000600 and 0x07000601); when not needed set to **NULL** (i.e. if numeric parameter values are used)

Returns:

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

BOOL **E7XX_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. 47, for a list of valid parameter numbers.

Notes:

To save the currently valid parameters to flash ROM, where they become the power-on defaults, you must use **E7XX_WPA** (p. 37). Parameter changes not saved with **E7XX_WPA** will be lost when the controller is powered off.

If the same designator has the same parameter number more than once, only the **last** value will be set. For example `E7XX_SPA(id, "111", {0x07000300, 0x07000300, 0x07000301}, {3e-2, 2e-2, 2e-4})` will set the P-term of '1' to 2e-2 and the I-term to 2e-4.

Arguments:

ID ID of controller

szAxes string with designators, one parameter is set for each designator in *szAxes*
for axis-related parameters: axis identifier;
for piezo- or sensor-related parameters: channel number;
otherwise a parameter-related code

piParameterArray Parameter numbers

pdValueArray array with the values for the respective parameters

szStrings string, with linefeed-separated parameter values (e.g. "X \nμm\n" are parameters 0x07000600 and 0x07000601); when not needed set to **NULL** (i.e. if numeric parameter values are used)

Returns:

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

BOOL **E7XX_STE** (int *ID*, const char *cAxis*, double *dStepSize*)

Corresponding command: STE

Record step response for one axis. The controller will move the given axis relative to the current position and record 8192 position values from start.

The number of servo cycles used for data recording depends on the setting made with **E7XX_RTR** (p. 32). Call **E7XX_qDRR_SYNC**() (p. 23) or **E7XX_qSTE**() (p.27) to read the recorded data.

Arguments:

ID ID of controller

cAxis axis for which the step response will be recorded

dStepSize size of step

Returns:

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

BOOL **E7XX_STP** (int *ID*)

Corresponding command: STP

Stops the motion of all axes instantaneously. Only non-complex motion (e.g. E7XX_MOV, E7XX_GOH, E7XX_SVR, E7XX_STE) can be interrupted with E7XX_STP. Error code 10 is set. After the stage was stopped, the target position is set to the current position.

Note: E7XX_HLT does not take effect to analog input which is used for "direct" axis control (see E-761 User Manual for more information). To disable "direct" control for an axis, the value of the corresponding "Aux-Input to target factor" parameter (ID 0x06000902) must be set to 0 with E7XX_SPA.

Arguments:

ID ID of controller

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **E7XX_SVA** (int *ID*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: SVA

Set absolute open-loop control value for ***szAxes***. Servo must be switched off when using this command.

The interpretation of the open-loop control value depends on the settings of the axis-to-PZT matrix (see "Output Generation" in the E-761 User Manual for more information). With the default matrix coefficients, open-loop control values numerically correspond to axis position values.

If the PZT control voltage resulting from the commanded open-loop control value exceeds the voltage limit of one of the PZT amplifiers which participate in this axis (see E7XX_VMA and E7XX_VMI), then the command is not executed (check with E7XX_GetError).

Arguments:

ID ID of controller
szAxes string with axes
pdValueArray open-loop control values for the axes

Returns:

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

BOOL **E7XX_SVO** (int *ID*, const char* *szAxes*, const BOOL* *pbValueArray*)

Corresponding command: SV0

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". When the servo is switched on, the target position is set to the current position. This avoids jumps when servo-control starts.

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. 7)

BOOL **E7XX_SVR** (int *ID*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: SVR

Set open-loop control values for ***szAxes*** relatively, i.e. increase last commanded open-loop control value by the specified values. Servo must be switched off when using this command.

The interpretation of the resulting open-loop control value depends on the settings of the axis-to-PZT matrix (see "Output Generation" in the E-761 User Manual for more information). With the default matrix coefficients, open-loop control values numerically correspond to axis position values.

If the PZT control voltage resulting from the commanded open-loop control value exceeds the voltage limit of one of the PZT amplifiers which participate in this axis (see E7XX_VMA and E7XX_VMI), then the command is not executed (check with E7XX_GetError)

Arguments:

ID ID of controller
szAxes string with axes
pdValueArray values to be added (algebraically) to open-loop control values of the affected axes

Returns:

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

BOOL **E7XX_VCO** (int *ID*, char* szAxes, const BOOL* *pbValueArray*)

Corresponding command: VC0

Set velocity-control "on" or "off". When velocity-control is "on", the corresponding axes will move with the currently valid velocity. That velocity can be set with **E7XX_VEL**() (*p. 36*).

Note:

During wave generator output, there is no velocity control, i.e. any settings made with **E7XX_VCO** will be ignored.

This function saves the parameters in RAM only. To save the currently valid parameters to flash ROM, where they become the power-on defaults, you must use **E7XX_WPA** (p. 37) without any parameter. Changes not saved with **E7XX_WPA** will be lost when the controller is powered off.

Arguments:

ID ID of controller
szAxes string with axes
pbValueArray modes for the specified axes, **TRUE** for "on", **FALSE** for "off"

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **E7XX_VEL** (int *ID*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: VEL

Set the velocities to use during moves of *szAxes* ("Servo Loop Slew Rate" parameter, ID 0x07000200).

This setting will be effective only when velocity control mode is ON for the specified axis (see **E7XX_VCO**). A range check is done—the velocity value must not be negative. The **E7XX_VEL** function saves the parameters in RAM only. To save the currently valid parameters to flash ROM, where they become the power-on defaults, you must use **E7XX_WPA** (p. 37). Changes not saved with **E7XX_WPA** will be lost when the controller is powered off.

Arguments:

ID ID of controller
szAxes string with axes
pdValueArray velocities for the axes

Returns:

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

BOOL **E7XX_VMA** (int *ID*, const char* *szPiezoChannels*, const double* *pdValueArray*)

Corresponding command: VMA

Set upper PZT voltage soft limit of given piezo channel (the "Output Voltage High Limit" parameter (ID 0x0C000001)).

A range check is done—the upper limit value must not be greater than the value of the "Max Voltage of Amplifier" parameter (ID 0x0B000008) and not be smaller than the "Min Voltage of Amplifier" parameter (ID 0x0B000007). You can query these limits with **E7XX_qSPA** (p. 27).

Make sure that the upper limit is not smaller than the lower PZT voltage soft limit set with **E7XX_VMI**.

Notes:

This function saves the parameters in RAM only. To save the currently valid parameters to flash ROM, where they become the power-on defaults, you must use **E7XX_WPA** (p. 37). Parameter changes not saved with **E7XX_WPA** will be lost when the controller is powered off.

On power-up or when rebooting the E-761, the upper PZT voltage soft limit is replaced by the value of the "Max Voltage of Amplifier" parameter, but can be restored to the saved value using **E7XX_RPA** (p. 32).

Arguments:

ID ID of controller
szPiezoChannels string with PZT channels
pdValueArray upper limits for PZT voltage

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **E7XX_VMI** (int *ID*, const char* *szPiezoChannels*, const double* *pdValueArray*)

Corresponding command: VMI

Set lower PZT voltage soft limit of given piezo channel (the "Output Voltage Low Limit" parameter (ID 0x0C000000)).

A range check is done—the lower limit value must not be smaller than the value of the "Min Voltage of Amplifier" parameter (ID 0x0B000007) and not be greater than the "Max Voltage of Amplifier" parameter (ID 0x0B000008). You can query these limits with **E7XX_qSPA** (p. 27).

Make sure that the lower limit is not greater than the upper PZT voltage soft limit set with **E7XX_VMA**.

Notes:

This function saves the parameters in RAM only. To save the currently valid parameters to flash ROM, where they become the power-on defaults, you must use **E7XX_WPA** (p. 37). Parameter changes not saved with **E7XX_WPA** will be lost when the controller is powered off.

On power-up or when rebooting the E-761, the lower PZT voltage soft limit is replaced by the value of the "Min Voltage of Amplifier" parameter, but can be restored to the saved value using **E7XX_RPA** (p. 32).

Arguments:

ID ID of controller
szPiezoChannels string with PZT channels
pdValueArray lower limits for PZT voltage

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **E7XX_VOL** (int *ID*, const char* *szPiezoChannels*, const double* *pdValueArray*)

Corresponding command: VOL

Set absolute PZT voltages for *szPiezoChannels*. Servo must be switched off when calling this function. If the commanded voltage exceeds the voltage limits of the PZT channel (see **E7XX_VMA** and **E7XX_VMI**), then the function is not executed (check with **E7XX_qERR**).

Arguments:

ID ID of controller
szPiezoChannels string with PZT channels
pdValueArray voltages for the PZT channels

Returns:

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

BOOL **E7XX_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 EPROM. 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. 47 for valid parameter numbers.

Notes:

CAUTION: If current parameter values are incorrect, the system may malfunction. Be sure that you have the correct parameter settings before using **E7XX_WPA**.

Settings not saved with **E7XX_WPA** will be lost when the controller is powered off or rebooted.

With **E7XX_qHPA** (p. 24) you can obtain a list of the parameters IDs.

Use **E7XX_qSPA** (p. 27) to check the current parameter settings in the volatile memory.

Parameters can be changed with **E7XX_SPA** (p. 34), **E7XX_AVG** (p. 17), **E7XX_DFH** (p. 18), **E7XX_RTR** (p. 32), **E7XX_VEL** (p. 36), **E7XX_VMA** (p. 36), **E7XX_VMI** (p. 36) and **E7XX_WGC** (p. 45).

When **E7XX_WPA** is used without specifying any parameters, all currently valid parameter values are saved, and additionally the following settings are saved too:
velocity control mode (**E7XX_VCO**, p. 35),
position limits (**E7XX_NLM**, p. 21, **E7XX_PLM**, p. 21).

Arguments:

ID ID of controller
szPassword There is a password required to set parameters in the EPROM . This password is “100”
szAxes string with designators. For each designator in *szAxes* one parameter value is copied.
for axis-related parameters: axis identifier;
for piezo- or sensor-related parameters: channel number;
otherwise a parameter-related code

piParameterArray Array with parameter numbers

Returns:

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

6.2. Wave Generator

The E-761 has a flexible wave generator with which you can define arbitrary waveforms for the motion of up to three axes. If the mechanics connected to the controller only requires three PZT output channels, a fourth waveform can be output on the fourth PZT channel (see the E-761 User Manual for details). The waveforms are stored in wave tables for later output.

The IDs of wave tables and wave generators have to correspond to the numerical indexes of the logical axes (e.g. if axis Z is the third axis and you want to address the corresponding wave table and wave generator, their IDs must be 3).

The waveform values are absolute values. In closed-loop operation (servo ON), they are interpreted as target positions in either case. In open-loop operation (servo OFF), the interpretation of the wave generator output depends on the settings of the axis-to-PZT matrix (see "Output Generation" in the E-761 User Manual for more information). By default, the matrix is set up so that commanded open-loop control values numerically correspond to axis position values.

To facilitate creation of the desired waveforms, a number of basic commands are available. The waveform can be made up by concatenating a number of “segments”.

Digital output synchronized with the wave generator output and hence with the axis motion is possible via the start options bit 3, bit 4 and bit 5 of the E7XX_WGO function. To make the digital output available outside of the PC, a trigger output bracket is required (included with E-761.3CT models; can be ordered separately as E-761.00T). The assignment of the three trigger output lines to the wave generators (and hence to the axes) is fixed: TrigOut1 belongs to wave generator 1 (axis 1), TrigOut2 belongs to wave generator 2 (axis 2), and TrigOut3 belongs to wave generator 3 (axis 3). With wave generator 4, no digital trigger output is possible. See the E-761 User Manual for pinout of the Digital Out socket.

6.2.1. Function Overview

- BOOL **E7XX_IsGeneratorRunning** (int *ID*, const char* *szWaveGeneratorIDs*, BOOL* *pbValueArray*)
- BOOL **E7XX_qGWD** (int *ID*, char *cWaveTableId*, int *iOffsetOfFirstPointInWaveTable*, int *iNumberOfValues*, double* *pdValueArray*) (p.**40**)
- BOOL **E7XX_qTLT** (int *ID*, int* *piDdlTables*)
- BOOL **E7XX_qTWG** (int *ID*, int* *piGenerator*)
- BOOL **E7XX_qWAV** (int *ID*, const char* *szWaveTableIds*, const int* *piParameterIdsArray*, double* *pdValueArray*)
- BOOL **E7XX_qWGC** (int *ID*, char* const *szWaveGeneratorIds*, int* *piValueArray*)
- BOOL **E7XX_qWGO** (int *ID*, const char**szWaveGeneratorIds*, int* *iStartModArray*)
- BOOL **E7XX_qWMS** (int *ID*, const char* *szWaveTableIds*, int* *piMaxWaveSize*)
- BOOL **E7XX_WAV_SINP** (int *ID*, const char* *szWaveTableId*, int *iOffsetOfFirstPointInWaveTable*, int *iNumberOfPoints*, int *iAddAppendWave*, int *iCenterPointOfWave*, double *dAmplitudeOfWave*, double *dOffsetOfWave*, int *iSegmentLength*) (p.**42**)
- BOOL **E7XX_WAV_LIN** (int *ID*, const char* *szWaveTableId*, int *iOffsetOfFirstPointInWaveTable*, int *iNumberOfPoints*, int *iAddAppendWave*, int *iNumberOfSpeedUpDownPointsInWave*, double *dAmplitudeOfWave*, double *dOffsetOfWave*, int *iSegmentLength*) (p.**42**)
- BOOL **E7XX_WAV_PNT** (int *ID*, const char* *szWaveTableId*, int *iOffsetOfFirstPointInWaveTable*, int *iNumberOfPoints*, int *iAddAppendWave*, double* *pdWavePoints*) (p.**43**)
- BOOL **E7XX_WAV_RAMP** (int *ID*, const char* *szWaveTableId*, int *iOffsetOfFirstPointInWaveTable*, int *iNumberOfPoints*, int *iAddAppendWave*, int *iCenterPointOfWave*, int *iNumberOfSpeedUpDownPointsInWave*, double *dAmplitudeOfWave*, double *dOffsetOfWave*, int *iSegmentLength*) (p.**44**)
- BOOL **E7XX_WCL** (int *ID*, const long *iWaveTableId*)
- BOOL **E7XX_WGC** (int *ID*, const char* *szWaveTableIds*, const int* *piNumberOfCyclesArray*)
- BOOL **E7XX_WGO** (int *ID*, const char* *szWaveGeneratorIds*, const int* *iStartModArray*)
- BOOL **E7XX_WGR** (int *ID*)

6.2.2. Function Documentation

BOOL **E7XX_IsGeneratorRunning** (const int *ID*, const char* *szWaveGeneratorIDs*, BOOL* *pbValueArray*)

Corresponding **command:** #9 (ASCII 9)

Check if *szAxes* are engaged in an unfinished wave generator move. Motion due to other commands is not accounted for. If TRUE for an axis, the corresponding element of the array will be set to **TRUE**, otherwise to **FALSE.** If no axes were specified, only one boolean value is set and it is placed in *pbValueArray[0]*: It is **TRUE** if at least one axis is TRUE, **FALSE** otherwise.

Arguments:

ID ID of controller

szWaveGeneratorIDs string with wave generators, if "" or **NULL** all wave generators are queried and a global result placed in *pbValueArray[0]*

pbValueArray array to receive status, TRUE for wave generator in progress, FALSE otherwise

Returns:

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

BOOL **E7XX_qGWD** (int *ID*, char *cWaveTableId*, int *iOffsetOfFirstPointInWaveTable*, int *iNumberOfValues*, double* *pdValueArray*)

Corresponding **command:** GWD?

Read the waveform associated with *cWaveTableId*.

Notes:

The following fact which affects only the response to the **E7XX_qGDW** query and not the waveform output by the wave generator: The content of a wave table is not completely erased when a new waveform is written to this table. Only the number of points given by the new waveform is written beginning with the first point in the table, but any subsequent data points will keep the old values from the former waveform. You can query the number of points belonging to the current valid waveform using **E7XX_qWAV** (p. 41).

Arguments:

ID ID of controller

cWaveTableId identifier for wave table

iOffsetOfFirstPointInWaveTable index of first point to be read

iNumberOfValues number of points to read

pdValueArray array to receive the wave form. (Caller must provide enough space to store *nLength* double values!)

Returns:

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

BOOL **E7XX_qTLT** (int *ID*, int* *piDdlTables*)

Corresponding **command:** TLT?

Get the number of DDL data tables.

Arguments:

ID ID of controller

piDdlTables pointer to receive the number of DDL data tables.

Returns:

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

BOOL **E7XX_qTWG** (int *ID*, int* *piGenerator*)

Corresponding **command:** TWG?

Get the number of wave generators.

Arguments:

ID ID of controller
piGenerator pointer to store the number of wave generators.

Returns:

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

BOOL **E7XX_qWAV** (int *ID*, const char* *szWaveTableIds*, int* *piParameterIdsArray*, double* *pdValueArray*)

Corresponding **command:** WAV?

Get the parameters for a defined waveform. For each desired parameter you must specify a wave table in *szWaveTableIds* and a parameter ID in the corresponding element of *iCmdarray*. The following parameter ID is valid:

1: Number of waveform points for currently defined wave.

Arguments:

ID ID of controller
szWaveTableIds string with wave tables IDs for which the parameter(s) should be read
piParameterIdsArray array with IDs of requested parameters
pdValueArray array to be filled with the values for the parameters

Returns:

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

BOOL **E7XX_qWGC** (int *ID*, char* const *szWaveGeneratorIds*, int* *piValueArray*)

Corresponding **command:** WGC?

Get the number of wave generator output cycles set by **E7XX_WGC (p. 45)**.

Arguments:

ID ID of controller
szWaveGeneratorIds string with wave tables
piValueArray array with number of cycles for each wave table in *szWaveTableIds*

Returns:

TRUE if no error, FALSE otherwise

BOOL **E7XX_qWGO** (int *ID*, const char**szWaveGeneratorIds*, int* *iStartModArray*)

Corresponding **command:** WGO?

Get the wave generator start mode set by **E7XX_WGO (p.** 45**)**.

Arguments:

ID ID of controller
szWaveGeneratorIds string with wave generators for which the start mode values will be read out
iStartModArray array with modes for each wave generator in *szWaveGeneratorIds*

Returns:

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

BOOL **E7XX_qWMS** (int *ID*, const char* *szWaveTableIds*, int* *piMaxWaveSize*)

Corresponding **command:** WMS?

Gets the maximum size of the wave storage for *szWaveTableIds*

Arguments:

ID ID of controller

szWaveTableIds string with wave tables, if "" or **NULL** all wave tables are queried.
piMaxWaveSize array to be filled with the maximum size of the wave storage for the corresponding wave table (number of points).

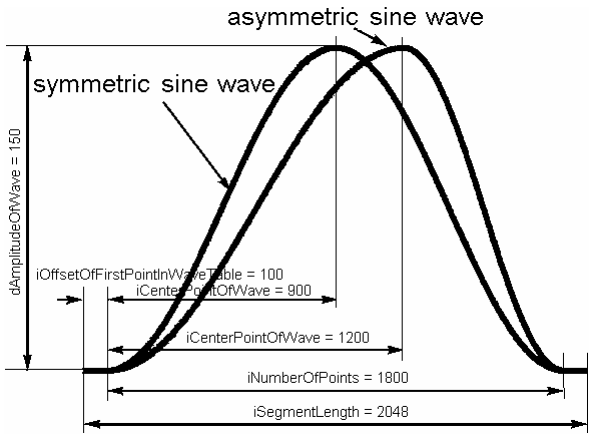
Returns:
TRUE if no error, FALSE otherwise (see p. 7)

BOOL **E7XX_WAV_SIN_P** (int *ID*, const char* *szWaveTableId*, int *iOffsetOfFirstPointInWaveTable*, int *iNumberOfPoints*, int *iAddAppendWave*, int *iCenterPointOfWave*, double *dAmplitudeOfWave*, double *dOffsetOfWave*, int *iSegmentLength*)

Corresponding **command:** `WAV SINP`

Produce a sine wave curve segment. The wave is not output at this time. 8192 points per wave table are available.

Note:
If the number of points is large, the calculation may take several seconds.



Arguments:
ID ID of controller
szWaveTableIds string with wave tables IDs
iOffsetOfFirstPointInWaveTable index of first point to be modified.
iNumberOfPoints number of points to modify
iAddAppendWave the following values are valid:
0 = the original wave curve segment is stored
1 = the wave curve segment is added to the last stored curve segments; not available for E-710
2 = the wave form will be appended to the last stored curve segment
iCenterPointOfWave the center point of the curve.
dAmplitudeOfWave the amplitude of the curve.
dOffsetOfWave the offset of the curve (see figure of **E7XX_WAV_LIN**() (*p.42*)).
iSegmentLength the length of the whole segment.

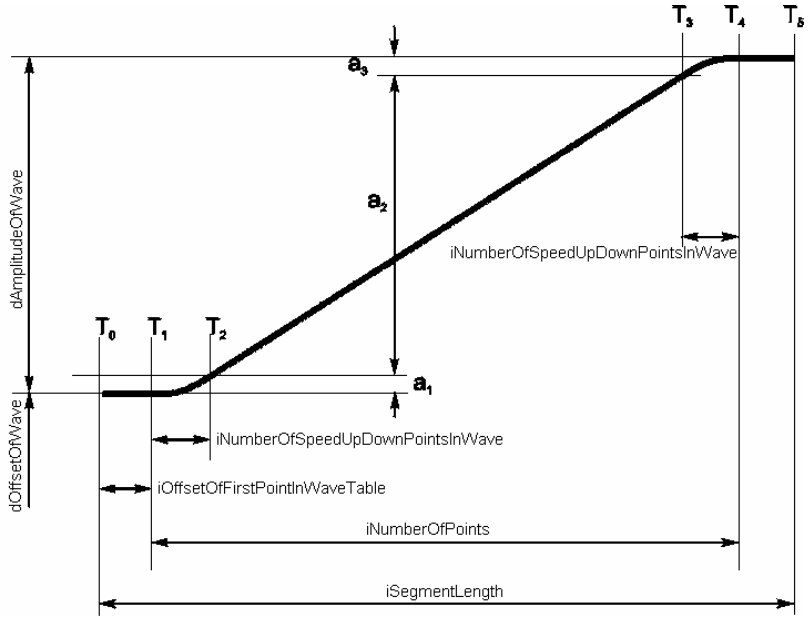
Returns:
TRUE if no error, FALSE otherwise (see p. 7)

BOOL **E7XX_WAV_LIN** (int *ID*, const char* *szWaveTableId*, int *iOffsetOfFirstPointInWaveTable*, int *iNumberOfPoints*, int *iAddAppendWave*, int *iNumberOfSpeedUpDownPointsInWave*, double *dAmplitudeOfWave*, double *dOffsetOfWave*, int *iSegmentLength*)

Corresponding **command:** `WAV LIN`

Have the wave generator produce a single line. The wave is not output at this time. 8192 points per wave table are available.

Note:
If the number of points is large, the calculation may take several seconds.



Arguments:

ID ID of controller
szWaveTableIds string with wave tables IDs
iOffsetOfFirstPointInWaveTable index of first point to be modified
iNumberOfPoints number of points to modify
iAddAppendWave the following values are valid:
0 = the original wave curve segment is stored
1 = the wave curve segment is added to the last stored curve segments; not available for E-710
2 = the wave form will be appended to the last stored curve segment
iNumberOfSpeedUpDownPointsInWave the size of the speed up and down
dAmplitudeOfWave the amplitude of the wave.
dOffsetOfWave the offset of the curve.
iSegmentLength the length of the whole segment.

Returns:

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

BOOL **E7XX_WAV_PNT** (int *ID*, const char* *szWaveTableId*, int *iOffsetOfFirstPointInWaveTable*, int *iNumberOfPoints*, int *iAddAppendWave*, double* *pdWavePoints*)

Corresponding **command:** `WAV PNT`

Downloads a user-defined wave to the E-7xx controller. The wave is not output at this time. 8192 points per wave table are available.

Note:

If the number of points is large, the calculation may take several seconds.

Arguments:

ID ID of controller
szWaveTableIds string with wave tables IDs
iOffsetOfFirstPointInWaveTable index of first point to be written. E-761: Starts with 0.
iNumberOfPoints number of points to be written
iAddAppendWave the following values are valid:
0 = the original wave curve segment is stored
1 = the wave curve segment is added to the last stored curve segments; not available for E-710
2 = the wave form will be appended to the last stored curve segment
pdWavePoints array with the wave points.

Returns:

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

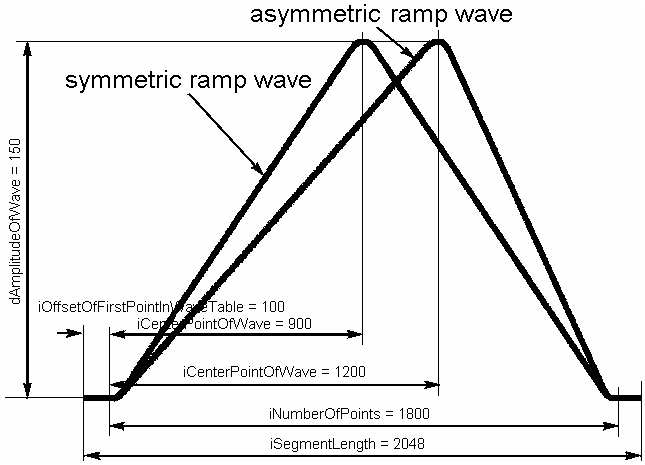
BOOL **E7XX_WAV_RAMP** (int *ID*, const char* *szWaveTableId*, int *iOffsetOfFirstPointInWaveTable*, int *iNumberOfPoints*, int *iAddAppendWave*, int *iCenterPointOfWave*, int *iNumberOfSpeedUpDownPointsInWave*, double *dAmplitudeOfWave*, double *dOffsetOfWave*, int *iSegmentLength*)

Corresponding **command:** `WAV RAMP`

Have the wave generator produce a ramp wave. The wave is not output at this time. 8192 points per wave table are available.

Note:

If the number of points is large, the calculation may take several seconds.



Arguments:

ID ID of controller
szWaveTableIds string with wave tables IDs
iOffsetOfFirstPointInWaveTable index of first point to be modified.
iNumberOfPoints number of points to modify
iAddAppendWave the following values are valid:
0 = the original wave curve segment is stored
1 = the wave curve segment is added to the last stored curve segments; not available for E-710
2 = the wave form will be appended to the last stored curve segment
iCenterPointOfWave the center point of the wave.
iNumberOfSpeedUpDownPointsInWave the size of the speed up and down (see figure of **E7XX_WAV_LIN**() (*p.**42***)).
dAmplitudeOfWave the amplitude of the wave.
dOffsetOfWave the offset of the curve (see figure of **E7XX_WAV_LIN**() (*p.**42***)).
iSegmentLength the length of the whole segment.

Returns:

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

BOOL **E7XX_WCL** (int *ID*, const long *iWaveTableId*)

Corresponding **command:** `WCL`

Clears waveform associated with specified wave table.

Arguments:

ID ID of controller
iWaveTableId ID of the wave table to be cleared.

Returns:

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

BOOL **E7XX_WGC** (int *ID*, const char* *szWaveTableIds*, int* *piNumberOfCyclesArray*)

Corresponding command: WGC

Set the number of cycles for the wave generator output ("Wave generator cycles" parameter, ID 0x13000003). The wave generator output is started with **E7XX_WGO**.

Note: E7XX_WGC saves the new value in RAM only. To save the currently valid value to non-volatile memory you must use E7XX_WPA. Changes not saved with E7XX_WPA will be lost when the controller is powered off.

Arguments:

ID ID of controller
szWaveTableIds string with wave tables
piNumberOfCyclesArray array with number of cycles for each wave table in *szWaveTableIds*

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **E7XX_WGO** (int ID, const char**szWaveGeneratorIds*, int* *iStartModArray*)

Corresponding **command:** WGO

Start or stop output of stored waveform and set wave generator output mode. The output mode is set with a separate bit mask for each wave generator (= axis). When no bits are set, there is no wave generator output for the corresponding axis. Each time the wave generator is started recording starts automatically. Read the data with **E7XX_qDRR_SYNC**, p. 23. Recording can be restarted with **E7XX_WGR** (p. 46).

Up to four wave generators can run simultaneously.

Note that bit 3 (0x8 or 8), bit 4 (0x10 or 16), bit 5 (0x20 or 32) and bit 8 (0x100 or 256) cannot start the wave generator output by themselves. They simply specify certain start options and must always be combined with one of the start modes specified in bit 0 (0x1 or 1), bit 1 (0x2 or 2), bit 2 (0x4 or 4) and bit 10 (0x400 or 1024). If you should combine bits 0, 1, 2 and 10, the wave generator starts with the mode given by the least significant bit.

Digital output synchronized with the wave generator output and hence with the axis motion is possible via the start options bit 3, bit 4 and bit 5. To make the digital output available outside of the PC, a trigger output bracket is required (included with E-761.3CT models; can be ordered separately as E-761.00T).

The number of output cycles can be set with **E7XX_WGC** (p. 45). Recording is pre-configured, see **E7XX_qDRR_SYNC**, p. 23. Recording always takes place for all record tables, regardless of which wave generator was started. Recording ends when the record table content has reached the maximum number of points (8192 per table). When the wave generator output is synchronized by interrupt (started with bit 0 or 1), the wave table rate (i.e. the output frequency) depends on the servo sampling rate influenced by **E7XX_AVG** (p. 17).

Arguments:

ID ID of controller
szWaveGeneratorIds string with wave generators.
iStartModArray array with modes for each wave generator in *szWaveGeneratorIds* (hex format, optional decimal format):

- 0: wave generator output is stopped
- bit 0: wave generator output started and synchronized by interrupt
- bit 1: wave generator output started and synchronized by interrupt and gated by external signal, The external signal used is the analog input signal (see E-761 User Manual). The wave generator runs as long as the signal is HIGH.
- bit 2: generator is started and synchronized by external signal. NOTE: The external signal used is the analog input signal (see E-761 User Manual). The wave generator runs as long as the signal is HIGH. The external signal (both high and low level) must have width of more than 50 microseconds.
- bit 3: synchronized trigger pulse is output on digital output line when the wave generator outputs a new data point; start option
- bit 4: synchronized trigger pulse is output on digital output line when the axis finishes each period (end of scan line); start option
- bit 5: synchronized trigger pulse is output on digital output line when the axis reaches the amplitude limit (scan field limit, see WAV CFG); start option

bit 8: wave generator started at the endpoint of the last cycle; start option. The second and all subsequent output cycles each start at the endpoint of the preceding cycle. The final position is the sum of the endpoint of the last output cycle and any offset defined with WAV for the waveform.

bit 10: "external wave generator" is started—i.e. the analog input is enabled for commanding the axis given by the wave generator ID (see E-761 User Manual for more information).

Returns:

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

BOOL **E7XX_WGR** (int *ID*)

Corresponding **command:** `WGR`

Starts a new recording when a wave generator is running. Data can be read with **E7XX_qDRR_SYNC** (p. 23).

Recording always takes place for all record tables, regardless of which wave generator is running. The assignment of axis and data sources to the data recorder tables is as follows:

table 1: axis 1 actual position
table 2: axis 2 actual position
table 3: axis 3 actual position
table 4: analog input voltage (same value as read with **E7XX_qTAV**, i.e. contains gain and offset for the analog input, see E-761 User Manual)

Recording starts always with the next start point of the waveform, i.e. there might be a short delay between sending E7XX_WGR and the start of the record. If more than one wave generator is running, recording starts at the waveform start point which occurs first.

Recording ends when the record table content has reached the maximum number of points (8192).

Arguments:

ID ID of controller

Returns:

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

7. System Parameter Overview

CAUTION

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

It is strongly recommended to save the parameter values of the E-761 to a file on the host PC before you make any changes. This way the original settings can be restored if the new parameter settings will not prove satisfactory. To save the parameter values and to load them back to the E-761, use the *Device Parameter Configuration* window of NanoCapture™. See "Creating Backup File for Controller Parameters" in the E-761 User Manual for more information.

To adapt the E-761 to your application, you can modify parameter values—either for the whole system, for the individual axes or for the individual sensor channels and PZT amplifier channels (for the interdependence between axis and channels see "Principle of Operation" in the E-761 User Manual). The parameters and parameter types available depend on the controller firmware. Note that many parameters are "protected" and can not be changed—it is only possible to change the parameters which are listed in the table below (can be queried with E7XX_qHPA).

NOTE

The parameters which are available in the controller—protected and unprotected—are listed in the *Device Parameter Configuration* window of *NanoCapture™*. The unprotected parameters have the value 0 in the *CCL* column.

Parameters can be changed temporarily or in non-volatile memory using the *Device Parameter Configuration* window of *NanoCapture™* (*Config → Device Parameter Configuration* menu sequence). Alternatively you can enter appropriate GCS commands in the command terminal (see E7XX_SPA, E7XX_SEP, E7XX_WPA), but using the *Device Parameter Configuration* window is much more comfortable because you do not have to deal with any parameter numbers. The parameters which can be changed have the value 0 in the *CCL* column of the *Device Parameter Configuration* window.

NOTE

See the Notes column in the list below for special parameter characteristics:

→ Parameters may be read-only even though their CCL value is 0

→ Parameters may only be present in volatile memory

→ Parameters may be modifiable only for a certain axis or channel

→ Parameters may refer to the whole system. For those parameters the ItemID in the appropriate commands must always have the value 1

See also the *NanoCapture*™ manual for how to edit, save or reset parameter values.

In addition to the "general" modification commands E7XX_SPA and E7XX_SEP, there are functions for commands which change certain specific parameters. All the functions listed below change the parameter value only in volatile memory, and E7XX_WPA must be used to save the value to non-volatile memory:

E7XX_AVG (p. 17; "Sensor sampling time" (ID 0x0e000100) and "Servo update time" (ID 0x0e000200))

E7XX_DFH (p. 18; "User Origin" (ID 0x07010200))

E7XX_RTR (p. 32; "Table rate" (ID 0x16000000))

E7XX_VEL (p. 36; “Servo loop slew rate” (ID 0x07000200))

E7XX_VMA (p. 36; "Output Voltage High Limit" (ID 0x0C000001))

E7XX_VMI (p. 36; "Output Voltage Low Limit" (ID 0x0C000000))

E7XX_WGC (p. 45; "Wave generator cycles" (ID 0x13000003))

Values stored in non-volatile memory are power-up defaults, so that the system can be used in the desired way immediately. Note that PI records data files of every E-761 controller calibrated at the factory for easy restoration of original settings after shipping.

Note that when a stage with ID-chip is connected to the controller for the first time, the stage parameters from the ID-chip will be written to the EEPROM on PC power-on (= controller power-on). Afterwards, the stage parameters will be written on power-on only when the "Read ID-Chip always" parameter is enabled, in this case the home-position is reset. By default, this option is disabled to maintain optimized parameter settings on the controller. The parameters which are stored on the ID-chip are marked in the Notes column of the table below, but can differ slightly between the different mechanics provided by PI. See "ID-Chip Support / Stage Replacment" in the E-761 User Manual for more information about the handling of stages with ID-chip.

Parameter Number	Parameter Name	Range	Notes
0X02000000	Sensor Mechanic: Sensor/Analog enable	0 = Disabled 1 = Enabled	ID-Chip
0X02000001	Sensor Mechanic: Sensor type		
0X02000100	Sensor Mechanic: Sensor range factor	0 = Board Range 3.00X 1 = Option 3.00X 21 2 = Option 3.00X 31 3 = Option 3.00X 41 4 = Option 3.00X 51 5 = Option 3.00X 61 6 = Option 3.00X 71 7 = Board Range 2.13X 8 = Option 2.13X 32 9 = Option 2.13X 42 10 = Option 2.13X 52 11 = Option 2.13X 62 12 = Option 2.13X 72 13 = Board Range 1.25X 14 = Option 1.25X 43 15 = Option 1.25X 53	ID-Chip

Parameter Number	Parameter Name	Range	Notes
		16 = Option 1.25X 63 17 = Option 1.25X 73 18 = Board Range 1.00X 19 = Option 1.00X 54 20 = Option 1.00X 64 21 = Option 1.00X 74 22 = Board Range 0.75X 23 = Option 0.75X 65 24 = Option 0.75X 75 25 = Board Range 0.68X 26 = Option 0.68X 76 27 = Board Range 0.56X	
0X02000101	Sensor Mechanic: Board Gain	0 = Gain 0.5 64 = Gain 1.0 128 = Gain 2.0 192 = Gain 3.0	ID-Chip
0X02000102	Sensor Mechanic: Electrical poti selected		
0X04000001	ADC: PGA correction of gain 1.0		Can only be changed for the analog input line which is handled as the 4th sensor (channel of the A/D converter), i.e. ItemID must be 4 for write operations
0X04000101	ADC: PGA correction offset x (x = index)		Can only be changed for the analog input which is handled as the 4th sensor (channel of the A/D converter), i.e. ItemID must be 4 for write operations
0X05000000	Sensor Filter: Digital filter type	0 = No Filter 1 = IIR Filter 2 = FIR filter	ID-Chip
0X05000001	Sensor Filter: Digital filter Bandwidth/Hz		ID-Chip
0X05000002	Sensor Filter: Digital filter order		
0X05000101	Sensor Filter: User filter parameter A0		
0X05000102	Sensor Filter: User filter parameter A1		
0X05000103	Sensor Filter: User filter parameter B0		
0X05000104	Sensor Filter: User filter parameter B1		
0X05000105	Sensor Filter: User filter parameter B2		

Parameter Number	Parameter Name	Range	Notes
0x06000902	Target Manipulation: Aux-Input to target factor		
0X07000000	Servo: Range min limit (µ)		ID-Chip
0X07000001	Servo: Range max limit (µ)		ID-Chip
0X07000200	Servo: Servo loop slew rate (axis unit/ms)		ID-Chip
0X07000300	Servo: Servo loop P-Term		ID-Chip
0X07000301	Servo: Servo loop I-Term		ID-Chip
0X07000500	Servo: Position from sensor 1		ID-Chip
0X07000501	Servo: Position from sensor 2		ID-Chip
0X07000502	Servo: Position from sensor 3		ID-Chip
0X07000600	Servo: Axis name		ID-Chip
0X07000601	Servo: Axis unit		ID-Chip
0X07000800	Servo: servo ON/OFF start up	0 = Disabled 1 = Enabled	ID-Chip
0X07000900	Servo: Tolerance		ID-Chip
0X07000A00	Servo: Auto-Zero driving low voltage (V)		ID-Chip
0X07000A01	Servo: Auto-Zero driving high voltage (V)		ID-Chip
0X07000A02	Servo: AutoZero voltage [V]		
0X07000C00	Servo: Default position		
0X07000C01	Servo: Default voltage		
0X07010200	Servo: User origin		
0X08000100	Servo output filter: Notch frequency of filter nr. 1		ID-Chip
0X08000101	Servo output filter: Notch frequency of filter nr. 2		ID-Chip
0X08000200	Servo output filter: Notch rejection of filter nr. 1		ID-Chip
0X08000201	Servo output filter: Notch rejection of filter nr. 2		ID-Chip
0X08000300	Servo output filter: Notch bandwidth of filter nr. 1		ID-Chip
0X08000301	Servo output filter: Notch bandwidth of filter nr. 2		ID-Chip
0X09000000	Output Matrix: Driving with piezo 1		ID-Chip
0X09000001	Output Matrix: Driving with piezo 2		ID-Chip
0X09000002	Output Matrix: Driving with piezo 3		ID-Chip

Parameter Number	Parameter Name	Range	Notes
0X09000003	Output Matrix: Driving with piezo 4		ID-Chip
0X0C000000	Piezo: Output voltage low limit (V)		ID-Chip
0X0C000001	Piezo: Output voltage high limit (V)		ID-Chip
0X0D000600	System Local: Device ID		ItemID = 1 only
0X0E000100	System Global: Sensor sampling time		ItemID = 1 only
0X0E000200	System Global: Servo update time		ItemID = 1 only
0X0E000A00	System Global: Min temperature		ItemID = 1 only
0X0E000A01	System Global: Max temperature		ItemID = 1 only
0X0F000000	System Mechanic: Read ID-Chip always	0 = Disabled 1 = Enabled	
0x13000001	Wave Generator: Installed wave form		read-only; only in volatile memory
0x13000002	Wave Generator: Connected axis		read-only only in volatile memory
0x13000003	Wave Generator: Wave generator cycles		
0x13000004	Wave Generator: Max Wave Points		read-only only in volatile memory
0x13000102	Wave Generator: Total wave form points		only in volatile memory
0x13000109	Wave Generator: Wave generator table rate		
0x1300010A	Wave Generator: Number of wave tables		read-only; only in volatile memory; ItemID = 1 only
0x1300010B	Wave Generator: Curve offset		
0X16000000	Data Record: Table rate		ItemID = 1 only

8. Error Codes

The error codes listed here are those of the *PI General Command Set.* As such, some are not relevant to E-7xx controllers 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
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, call 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
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

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 while axis in motion, call CLR to resume operation
-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.

9. Index