Piezo · Nano · Positioning

PZ147E Software Manual

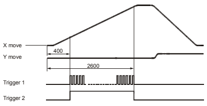
E-710 Windows GCS-DLL

(E7XX_GCS_DLL)

Release: 1.1.5 Date: 2007-01-11

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

- E-710 Digital Piezo Controller;
  3- and 4-axis versions firmware
  rev. 5.025/6.025 or newer and 7.xxx,
  6-axis version firmware
  rev. 2.12 or newer



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 2007-01-11
Document Number PZ147E, Release 1.1.5
E-710_GCSDLL_PZ147E.doc

Subject to change without notice. This manual is superseded by any new release. The newest release is avialable for download at www.pi.ws.

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_DLL

The E-7xx_GCS_DLL 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.

1.1. Quick Start

1.1.1. Installation

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.
- Follow the on-screen instructions. You can choose between “typical” and “custom” installation. Typical components are LabView drivers, DLLs, NanoCapture™, PZTControl, WinTerm32 and the manuals. “Typical” is recommended.

See Section 2.2 starting on p. 6 for more information about PI DLLs.

1.1.2. Connect the Controller

Connection to each E-710 can be made with RS-232 or with a National Instruments’ GPIB (IEEE 488) board. See the E-710 User Manual PZ 80E for more information. Note that the E7XX_GCS_DLL can be used with models with the PIO feature, but does not include PIO support.

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 *E710QuickTest.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.

Section 3.2 on p. 9 gives source code examples for axis initialization. See Section 5.1 on p. 14 for some simple operations with GCS commands and Section 5.2 on p. 35 for how to use the wave generators and the DDL functionality.

1.2. Non-GCS Firmware

It is possible to operate the E-710 with two command sets: the native ASCII command and the PI General Command Set. The native ASCII command set is understood by the E-710 firmware (see E-710 User Manual for more information). To enable the usage of the GCS commands with E-710, the E7XX_GCS_DLL.dll translates GCS commands into the native commands. Once the library is installed, you can use, for example, the LabVIEW GCS drivers to control the E-710 controller as though it were any GCS-compatible controller.

If you are using LabView, please read the documentation for the LabVIEW drivers to find out how to "connect" to the GCS library.

NOTES

Do not mix up the GCS command set and the native command set! GCS move commands do not work properly anymore after the position was changed by native commands.

Due to the emulation of the native command set, the execution of the E7XX_MOV and E7XX_MVR motion functions is noticeably slower than that of the native commands. Therefore the special non-GCS motion functions E7XX_NMOV and E7XX_NMVR are provided for applications which require quickest possible response to motion commands. See the function reference for details.

Although both command sets comprise the complete E-710 functionality, GCS commands can not be translated one-to-one into native commands. This is why there is no comparison of both command sets provided in this documentation.

1.3. Units and GCS

1.3.1. Conversion of Units

The GCS system uses physical units of measure. With some controllers, a scale factor can be applied, making a second physical unit (working unit) available without overwriting the default settings (see the **E7XX_DFF**()**, E7XX_qDFF**() function calls).

1.3.2. 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.

1.4. 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.

1.4.1. 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.

1.4.2. 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

1.4.3. Axis Renaming

The GCS DLL supports an axis-renaming scheme. See the **E7XX_SAI** function (p. 31) for details.

Keep in mind the following when dealing with axis names:

With E-710 controllers, the names displayed by the *NanoCapture*™ software may be different from the axis identifiers used in the function calls because *NanoCapture*™ permits independent assignment of axis names (maximal 4 characters).

1.5. About this Manual

Stages, Axes, and Channels (p. 2) explains the usage of the "axis", "channel" and "stage" terms and describes some configuration basics.
DLL Handling (p. 6) explains how to load the library and how to access the functions provided by the E7XX_GCS_DLL DLL.
Function Calls (p. 7) and Types Used in PI Software (p. 8) provide some general information about the syntax of most commands in the DLL.
Controller Setup (p. 9) describes the steps necessary at startup of the library.
Communication Initialization (p. 11) shows how to initiate communication with an E-7xx controller (see also Interface Settings (p.13)).
Functions for GCS Commands (p. 14) describes the functions encapsulating the embedded commands of the E-710: while Motion and Controller Configuration (p. 14) lists all functions required for "normal" operation, Wave Generator and DDL (p. 35) describes the wave generator usage and the appropriate functions for the E-7xx.
System Parameter Overview (p. 48) lists axis-, channel-, and system-specific parameters which can be set using several DLL functions.
Error Codes (p. 53) has a description of the possible errors.

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. 13). The error codes and messages are listed in “Error Codes” (p. 53).

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.25, **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. See also "System Parameter Overview" on p. 48.

Call **E7XX_SPA**() (p. 32) to modify parameters temporarily (to save them to EPROM use **E7XX_WPA**, p. 33), or **E7XX_SEP** (p. 31) to change the EPROM values.

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-710, and (without the call printf()) represents a typical initialization. In the example, **E7XX_qSAI**() (p. 26) is called to get the configured axes, and then **E7XX_INI**() (p. 21) is called to initialize these axes.

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

// connect to the E-710 (4 channel) over RS-232 (COM port 1, baudrate 9600)
ID = E7XX_ConnectRS232 (1, 9600);
if (ID<0)
    return FALSE;

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

// unconfigure axes 3 and 4
if(!E7XX_CST(ID, “34”, “NOSTAGE \nNOSTAGE\n”)
     return FALSE;

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

// call INI for all axes
// "" as axes string will address all configured axes
if (!E7XX_INI(ID, ""))
    return FALSE;
```

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-710 used with single-axis stages.

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

```
// connect to the E-710 over GPIB (board number 0, device address 4)
ID = E7XX_ConnectNIgpib(0, 4);
if (ID<0)
    return FALSE;

if (!E7XX_qCST(ID, "1234", stages, 1023))
    return FALSE;

// If all axes are configured,
// the output should be "1=ID-STAGE \n2=ID-STAGE \n3=ID-STAGE \n4=ID-STAGE\n"
printf("qCST() returned \"%s\"", stages);

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

// The output should be "1234" - axes 1, 2, 3 and 4 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 and 4 to NOSTAGE.
sprintf(stages, " NOSTAGE \n NOSTAGE ");
if (!E7XX_CST(ID, "34", stages))
    return FALSE;

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

// The output should be "12" - the new configured axes (axis 3 and 4 are now non-configured).
printf("qSAI() returned \"%s\"", axes);

if (!E7XX_qCST(ID, "1234", stages, 1023))
    return FALSE;

// The output should be "1=ID-STAGE \n2=IDSTAGE \n3=NOSTAGE \n4=NOSTAGE\n"
printf("qCST() returned \"%s\"", stages);

// call INI for all axes
// "" as axes string will address all configured axes
if (!E7XX_INI(ID, ""))
    return FALSE;
```

4. Communication Initialization

4.1. Functions

- BOOL **E7XX_ChangeNIgpibAddress** (int *ID*, int *iDeviceAddress*)
- int **E7XX_ConnectRS232** (int *iPortNumber*, int *iBaudRate*)
- int **E7XX_ConnectNIgpib** (int *iBoardNumber*, int *iDeviceAddress*)
- int **E7XX_InterfaceSetupDlg** (const char* *szRegKeyName*)
- 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_InterfaceSetupDlg**() , **E7XX_ConnectNIgpib**(), **E7XX_ConnectRS232**(), **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

BOOL **E7XX_ChangeNIgpibAddress** (int *ID*, int *iDeviceAddress*)

Change the IEEE488 address of an E-7xx Controller.

Arguments:

iDeviceAddress address of connected device

Returns:

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

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

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

Arguments:

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

int **E7XX_ConnectNIgpib** (int *iBoardNumber*, const long *iDeviceAddress*)

Open a connection from a National Instruments IEEE 488 board to an E-7xx. All future calls to control this E-7xx need the ID returned by this call.

Arguments:

iBoardNumber number of board (check with NI installation software)

iDeviceAddress address of connected device

Returns:

ID of new object, **-1** if interface could not be opened or no E-710 is responding.

int **E7XX_ConnectRS232** (int *iPortNumber*, const long *iBaudRate*)

Open an RS-232 ("COM") interface to an E-7xx. The call also sets the baud rate on the controller side. All future calls to control this E-7xx need the ID returned by this call.

Arguments:

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

Returns:

ID of new object, **-1** if interface could not be opened or no E-710 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.53) and retuned. After this call the DLL internal error state will be cleared; the controller error state will be cleared if it was queried.

Returns:

error ID, see **Error codes** (p.**53**) for the meaning of the codes.

int **E7XX_InterfaceSetupDlg** (const char* *szRegKeyName*)

Open dialog to let user select the interface and create a new E7xx object. All future calls to control this E-7xx need the ID returned by this call. See **Interface Settings** (p. 13) for a detailed description of the dialogs shown.

Arguments:

szRegKeyName key in the Windows registry in which to store the settings, the key used is `"HKEY_LOCAL_MACHINE\SOFTWARE\<your keyname>"` if *keyname* is **NULL** or "" the default key `"HKEY_LOCAL_MACHINE\SOFTWARE\PI\E7XX_GCS_DLL"` is used.

Note:

If your programming language is C or C++, use '\\' if you want to create a key and a subkey at once. To create `"MyCompany\E7XX_DLL"` you must call

```
E7XX_InterfaceSetupDlg( "MyCompany\\E7XX_GCS_DLL" )
```

Returns:

ID of new object, **-1** if user pressed "CANCEL", the interface could not be opened, or no E-710 is responding.

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

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

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.

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.

Arguments:

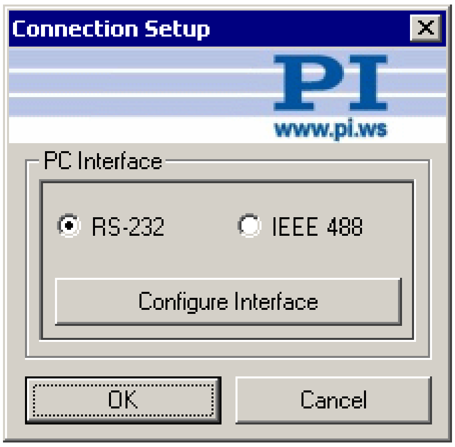
iErrorNumber number of error, as returned from **E7XX_GetError**()(p.12).
szErrorMessage pointer to buffer that will store the message
iBufferSize size of the buffer

Returns:

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

4.4. Interface Settings

When the interface setup dialog is shown (due to a call of **E7XX_InterfaceSetupDlg**, p. 12), the user has the choice between RS-232 and IEEE 488 (currently only National Instruments IEEE boards are supported).



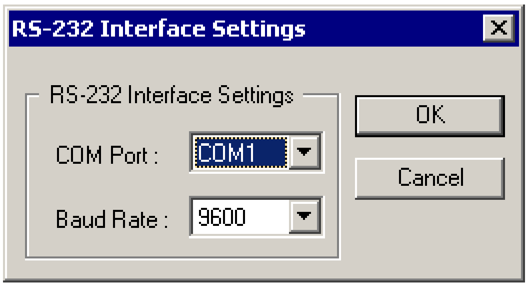
Connection Setup

PC Interface

● RS-232 ○ IEEE 488

4.4.1. RS-232 Settings

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



RS-232 Interface Settings

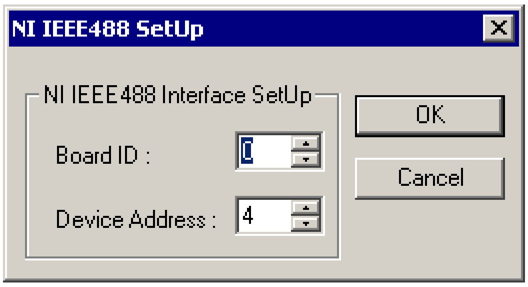
RS-232 Interface Settings

COM Port : COM1

Baud Rate : 9600

4.4.2. IEEE488 Settings

- `Board ID`: ID of the National Instruments board installed. If only one board is installed this will be 0, as in the most cases. Use the National Instruments setup and test software to determine the board ID.



NI IEEE488 SetUp

NI IEEE488 Interface SetUp

Board ID : 0

Device Address : 4

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

5. Functions for GCS Commands

To enable the usage of the GCS commands with E-710, the E7XX_GCS_DLL.dll translates GCS commands into the native commands which are understood by the E-710 firmware. The appropriate DLL functions are described in this Section.

You can send the commands whose functions are listed here directly—either using the E7XX_GcsCommandset function or the terminal in a GCS-based program (*NanoCapture*™ or *PZTControl)*. For the syntax see Section 5.3. Note that it is not possible to send GCS commands to the E-710 using a simple terminal program like *WinTerm32*.

5.1. Motion and Controller Configuration

The following examples are based on GCS commands, but the corresponding DLL functions can be used accordingly.

Examples regarding the wave generator and DDL usage can be found in Section 5.2 on p. 35.

5.1.1. Example: How to Rename an Axis

Action	Content of Program Window	Comment
Send:	SAI? ALL	Check the axis names of all available axes
Response:	1 2 3	
Send:	SAI 1 X	Rename axis "1" to "X"; note that names are limited to one character, case insensitive

5.1.2. Example: How to Command a Voltage to an Axis

In the following example, only piezo channel 1 participates in the motion of axis X. If multiple piezo channels would participate in axis' X motion, the axis' voltage probably would differ from the piezo channel voltage due to the axis-to-piezo-channel transformation (see E-710 User Manual for more information).

Action	Content of Program Window	Comment
Send:	SVO X 0	Servo should be OFF for axis X
Send:	SPA? 1 0x0C000000 1 0x0C000001	Check the hardware output range of piezo channel 1
Response:	1 0X0C000000=-2.00000000e+1 1 0X0C000001=1.20000000e+2	The hardware output range is -20 V to + 120 V.
Send:	SVA X 20	Set 20 V to axis X; the commanded value should be inside the limit range
Send:	SVR X 1.0	Increase voltage for axis X by 1 V
Send:	VOL? 1	Ask the voltage of piezo channel 1
Response:	1=21.0	

5.1.3. Example: How to Command a Position to an Axis

Action	Content of Program Window	Comment
Send:	SVO X 1	Servo should be ON for axis X
Send:	TMN? X	Get the low end of the moving range of axis X
Response:	X=0.00000000e+0	
Send:	TMX? X	Get the high end of the moving range of axis X

Response:	X=1.00000000e+2	Axis X can move from 0 to 100 µm.
Send:	MOV X 10	Move axis X to 10 µm.
Send:	MVR X 1.0	Increase the position of axis X by 1.0 µm
Send:	POS? X	Ask the current position of axis X
Response:	X=11.0	

5.1.4. Example: How to set a Velocity for an Axis

Action	Content of Program Window	Comment
Send:	SPA? X 0x07000200	Check the slew-rate of axis X
Response:	X 0x07000200=10.0	This means that the maximum velocity of axis X is 10 µm/ms (change the parameter setting e.g. with SPA to modify the maximum velocity)
Send:	VEL X 0.01	Set the velocity of axis X to 0.01 µm/ms = 10 µm/s; the commanded velocity should not exceed the maximum value given by the slew-rate parameter
Send:	MOV X 0	Move axis X to 0 µm.

5.1.5. Functions

- BOOL **E7XX_7XXReadLine** (int *ID*, char* *szStringr*, int *iBufferSize*)
- BOOL **E7XX_7XXSendString** (int *ID*, const char* *szCommand*)
- BOOL **E7XX_ATZ** (int *ID*, const char* *szAxes*, const double* *pdLowVoltageArray*, const BOOL* *pfUseDefaultArray*)
- BOOL **E7XX_CCL** (int *ID*, int *iCommandLevel*, const char**szPassWord*)
- BOOL **E7XX_CST** (int *ID*, const char* *szAxes*, const char* *szNames*)
- BOOL **E7XX_DFF** (int *ID*, const char* *szAxes*, const double* *pdValueArray*)
- BOOL **E7XX_DFH** (int *ID*, const char* *szAxes*)
- BOOL **E7XX_DPO** (int *ID*, const char* *szAxes*)
- BOOL **E7XX_DRC** (int *ID*, const int* *piRecordChannelIdsArray*, const char**szRecordSourceIds*, const int* *piRecordOptionArray*, const int* *piTriggerOptionArray*)
- 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_INI** (int *ID*, const char* *szAxes*)
- BOOL **E7XX_MOV** (int *ID*, const char* *szAxes*, const double* *pdValueArray*)
- BOOL **E7XX_MVR** (int *ID*, const char* *szAxes*, const double* *pdValueArray*)
- BOOL **E7XX_NMOV** (int *ID*, const char* *szAxes*, const double* *pdValueArray*)
- BOOL **E7XX_NMVR** (int *ID*, const char* *szAxes*, const double* *pdValueArray*)
- BOOL **E7XX_qCCL** (int *ID*, int* *piComandLevel*)
- BOOL **E7XX_qCST** (int *ID*, const char* *szAxes*, char* *szNames*, int *iBufferSize*)
- BOOL **E7XX_qDFF** (int *ID*, const char* *szAxes*, double* *pdValueArray*)
- BOOL **E7XX_qDFH** (int *ID*, const char* *szAxes*, double* *pdValueArray*)
- BOOL **E7XX_qDRC** (int *ID*, const int* *piRecordChannelIdsArray*, char* *szRecordSourceIds*, int* *piRecordOptionArray*, int* *piTriggerOptionArray*, int *iArraySize*)
- 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_qONT** (int *ID*, const char* *szAxes*, BOOL* *piValueArray*)
- BOOL **E7XX_qPOS** (int *ID*, const char* *szAxes*, double* *pdValueArray*)
- 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_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_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_qVEL** (int *ID*, const char* *szAxes*, double* *pdValueArray*)
- BOOL **E7XX_qVOL** (int *ID*, const char* *szPiezoChannels*, double* *pdValueArray*)
- BOOL **E7XX_qVST** (int *ID*, char* *szValideStages*, int *iBufferSize*)
- BOOL **E7XX_RPA** (int *ID*, const char* *szAxes*, const long* *iParameterArray*)
- 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_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_VEL** (int *ID*, const char* *szAxes*, const double* *pdValueArray*)
- BOOL **E7XX_WPA** (int *ID*, const char**szPassword*, const char* *szAxes*, const long* *iParameterArray*)

5.1.6. Function Documentation

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

For controllers with non-GCS firmware, the following functions encapsulate the embedded commands of the controller and provide some "shortcuts" to make the work easier.

BOOL **E7XX_E7XXReadLine** (int *ID*, char* *szString*, int *iBufferSize*)

Gets the answer to a native command of the E-710, provided its length does not exceed *iBufferSize*. The answers to a native command are stored inside the DLL, where as much space as necessary is obtained. Each call to this function returns and deletes the oldest answer in the DLL.

Note: See the E-710 User Manual for a description of the native commands which are understood by the E-710 firmware, and for a command reference.

Arguments:

ID ID of controller
szString the buffer to receive the answer.
iBufferSize the size of *szAnswer*.

Returns:

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

BOOL **E7XX_E7XXSendString** (int *ID*, const char* *szCommand*)

Sends a native command to the E-710. Any native command can be sent—this function is also intended to allow use of native commands not having a corresponding GCS function in the current version of the library.

Notes:

Do not mix up the GCS command set and the native command set! GCS move commands do not work properly anymore after the position was changed by native commands.

See the E-710 User Manual for a description of the native commands which are understood by the E-710 firmware, and for a command reference.

Arguments:

ID ID of controller

szCommand the GCS command as string.

Returns:

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

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.

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_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_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. 30). 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.

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_DFF** (int *ID*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding **command:** `DFF`

Defines scale *factor* which is applied to the basic unit (default is 1). E.g. 25000.4 changes the physical unit from µm to inches.

Example: The physical unit is µm and the scale factor is 1. The current position of an axis is 12. Now the scale factor is set to 3 with DFF. Reading the position gives 4 as result. A relative move of 1.5 causes the axis to move 4.5 µm.

Arguments:

ID ID of controller
szAxes string with axes.
pdValueArray scale factor, can only be positive

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

Corresponding **command:** `DPO`

DDL processing parameter correction for specified axis.

This function is available in command level 1 only (see **E7XX_CCL** (p. 17) and **E7XX_qCCL**).

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_DRC** (int *ID*, const int* *piRecordChannelIdsArray*, const const char**szRecordSourceIds*, cosnt int* *piRecordOptionArray*, const int* *PiTriggerOptionArray*)

Corresponding **command:** DRC

Configures the data recording which can be started with **E7XX_WGO** and **E7XX_WGR.** There are 6 record tables with 8192 points per table.

Notes:

For the recording which is started automatically with **E7XX_WGO**, you can configure the record tables for the moving axes according to the following rules:

- Only target position or actual position or position error can be recorded (*piRecordOptionArray*).
- An axis (*szRecordSourceIds*) can be assigned to only one record table. If an axis is assigned to more than one table, only the data for the last table connected to that axis will be recorded.

If you want to start recording with **E7XX_WGR**, configure the record tables according to the following rules:

- Do not change the default assignment of axes to record tables—axis 1 is connected to record table 1, axis 2 to record table 2 , …, axis 6 to record table 6.
- The data type (*piRecordOptionArray*) to be recorded must be the same for all record tables.
- To change the data type to be recorded for all tables, set the new data type for table 1 only. Note: If the change is done for a table other than table 1, the change will be ignored.

If no configuration is done with **E7XX_DRC**, the target positions will be recorded by default.

Arguments:

ID ID of controller
piRecordChannelIdsArray Id of the record table
szRecordSourceIds Id of the record source (axis)
piRecordOptionArray Can be one of:
0: The target position of the record source will be recorded.
1: The target position of the record source will be recorded.
2: The actual position of the record source will be recorded.
3: The position error of the record source will be recorded.
4: The DDL data of the record source will be recorded.
5: The driving voltage of the record source will be recorded.
piTriggerOptionArray Reserved.

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.

Only commands which have a function in the library can be sent.

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 DLL, where as much space as necessary is obtained. Each call to this function returns and deletes the oldest answer in the DLL.

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 DLL.

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 HLT. Error code 10 is set. After the stage was stopped, the target position is set to the current position.

Arguments:

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

Returns:

TRUE if no error, FALSE otherwise (see p. 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. Call **E7XX_qIMP**() (p.25) to read them.

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. Call **E7XX_qIMP**() (p.25) to read them.

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

Corresponding command: `INI`

Initialize *szAxes*. Stops the wave generator(s).

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_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_NMOV** (int *ID*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding **command:** `NMOV`

Caution: The use of **NMOV** is the same as that of the GCS command **MOV**, but it is not a GCS command!

Move *szAxes* to specified absolute positions.

This function is faster than **E7XX_MOV** (p. 21) but

- does not check range limits and servo states
- does not move the axes synchronously.

When the commanded target is outside the range limits, the axis will stop at its physical limit. To go back to normal operation, command the axis to a valid position using **E7XX_MOV**.

When servo is off, the axis does not move.

The commanded target can be queried with **E7XX_qMOV** (p. 25).

Note:

Due to the emulation of the E-710 native command set, the execution of E7XX_MOV is noticeably slower than those of the native commands. Therefore E7XX_NMOV is provided for applications which require quickest possible response to motion commands.

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_NMVR** (int *ID*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding **command:** `NMVR`

Caution: The use of **NMVR** is the same as that of the GCS command **MVR**, but it is not a GCS command!

Move *szAxes* relative to current target position.

This function is faster than **E7XX_MVR** (p. 21) but

- does not check range limits and servo states
- does not move the axes synchronously.

When the commanded target is outside the range limits, the axis will stop at its physical limit. To go back to normal operation, command the axis to a valid position using **E7XX_MOV**.

When servo is off, the axis does not move.

The commanded target can be queried with **E7XX_qMOV** (p. 25).

Note:

Due to the emulation of the E-710 native command set, the execution of E7XX_MVR is noticeably slower than those of the native commands. Therefore E7XX_NMVR is provided for applications which require quickest possible response to motion commands.

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_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_qDFF** (int *ID*, const char* *szAxes*, double* *pdValueArray)*

Corresponding **command:** `DFF?`

Returns constant unit value for specified axes (e.g. 25000.4 for inches).

Arguments:

ID ID of controller
szAxes string with axes, if "" or **NULL** all axes are affected.
pdValueArray array to receive the scale factor

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_qDRC** (int *ID*, const int* *piRecordChannelIdsArray*, char* *szRecordSourceIds*, int* *piRecordOptionArray*, int* *piTriggerOptionArray*, int *iArraySize*)

Corresponding **command:** `DRC?`

Returns the data recording configuration for the queried record table.

Arguments:

ID ID of controller
piRecordChannelIdsArray Id of the record table.
szRecordSourceIds array to receive the record source.
piRecordOptionArray array to receive the record option. The received value can be one of:

0: The target position of the record source will be recorded.
1: The target position of the record source will be recorded.
2: The actual position of the record source will be recorded.
3: The position error of the record source will be recorded.
4: The DDL data of the record source will be recorded.
5: The driving voltage of the record source will be recorded.

piTriggerOptionArray Reserved.
iArraySize array size for *piRecordChannelIdsArray*, *szRecordSourceIds*, *piRecordOptionArray* and *PiTriggerOptionArray*

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 starts either automatically when the wave generator is started with **E7XX_WGO** (p. 45), or can be started "manually" with **E7XX_WGR** (p. 45).

For the recorder configuration see **E7XX_DRC** (p. 19).

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 DLL 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. 53.

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. 48, for a list of valid parameter numbers for command level 1 (includes the parameter numbers valid for command level 0).

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.20) must have been called to run and record the step response

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_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_qPOS** (int *ID*, const char* *szAxes*, double* *pdValueArray*)

Corresponding command: `POS?`

Get the current positions of *szAxes*.

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_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 confuse with the "axis names" maintained by the controller (see Section 1.4.3 on p. 5).

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 confuse with the "axis names" maintained by the controller (see Section 1.4.3 on p. 5).

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. 48, 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. 48, 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_qSTE** (int *ID*, const char *cAxis*, int *iOffset*, int *nrValues*, double* *pdValueArray*)

Corresponding command: STE?

Get the recorded positions of a step response. The controller will move the given axis to the target position and record 8192 position values from start. Call **E7XX_STE**() (p.27) to start the step response.

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 PZT voltages for *szAxes* (see also **E7XX_qVOL**, p. 30 and footnote on this page[*]). If the corresponding command specified an out-of-range voltage, the limit is reported.

Arguments:

ID ID of controller

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

pdValueArray array to be filled with the voltage values for the axes

Returns:

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

[*] The voltages set and read for the axes may differ from those on the PZT channels because of the coordinate transformation performed. This is particularly true in the case of rotation axes, or if axes and channels are not parallel.

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.

Note that this function is available for 4-channel versions only and in command level 1 only (see **E7XX_CCL** on p. 17).

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_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.

This function is available in command level 1 only (see **E7XX_CCL** on p. 17)

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.

This function is available in command level 1 only (see **E7XX_CCL** on p. 17)

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.

Do not confuse with the “axis names” maintained by the controller (see Section 1.4.3 on p. 5).

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_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_qVOL** (int *ID*, const char* *szPiezoChannels*, double* *pdValueArray*)

Corresponding command: VOL?

Get current PZT voltages for *szPiezoChannels* (see also **E7XX_qSVA**, p.27 and footnote p. 27)

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_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. 48, 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_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.29). If the same axis identifier occurs more than once in *szOldAxes,* only the **last** occurrence will be used to change the name.

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. 48, for a list valid parameter numbers.

Warning! According to the non-volatile parameter memory chip manufacturer's specifications, only a few thousand write cycles can be executed successfully. Frequent re-configuring should therefore be avoided.

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. This function is only available in command level 1. Use **E7XX_CCL** (p.17) to change the command level. **E7XX_SEP** writes the parameters in EPROM and also in volatile memory (RAM).

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. 48, 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. 33). 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. Call **E7XX_qSTE**() (p.27) to read them.

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_SVA** (int *ID*, const char* *szAxes*, const double* *pdValueArray*)

Corresponding command: SVA

Set PZT voltages for *szAxes* to absolute values (see footnote p. 27). Servo must be switched off when using this command. The voltage specified for an axis is set on the PZT that is most-closely coupled to that axis. The other affected axes, if any, see voltage contributions in the proportions specified by the axis-to-PZT transformation matrix, resulting in (open-loop) motion in the direction of the specified axis. This command can be used to find defects or improper settings.

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray voltages 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: `SVO`

Set servo-control "on" or "off" (closed-loop/open-loop mode). If *pbValueArray[index]* is **FALSE** the mode is "off", if **TRUE** it is set to "on". 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 PZT voltages for *szAxes* relatively, i.e. increase last commanded voltages by the specified values (see footnote p. 27). Servo must be switched off when using this command.

If the axis voltage is out of range, the limit value is used, and the appropriate error flag is set.

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray values to be added (algebraically) to voltages of the affected axes

Returns:

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

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

Corresponding command: `VEL`

Set the maximum velocities to use during moves of *szAxes*.

Arguments:

ID ID of controller

szAxes string with axes

pdValueArray maximum velocities for the axes

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. 48 for valid parameter numbers.

Warning! According to the non-volatile parameter memory chip manufacturer's specifications, only a few thousand write cycles can be executed successfully. Frequent re-configuring should therefore be avoided.

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. 26) to check the current parameter settings in the volatile memory.

This function is only available in command level 1. Use **E7XX_CCL** (p.17) to change the command level.

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)

5.2. Wave Generator and DDL

5.2.1. Wave Generator Basics

With the E-710, it is possible to create arbitrary waveforms and to output them for up to two axes (E-710 is equipped with two wave generators).

The waveforms are stored in wave tables. Each axis has its own wave table. To address a wave table, the appropriate axis ID must be used (e.g. if you want to address the wave table of axis Z, the wave table ID must be Z). The total number of points available for all wave tables is 63488 (62464 with 6-axis versions). These points can be flexible assigned to the wave tables (see E7XX_WMS, p. 41), but due to memory restrictions, the number of tables to which the points are distributed must not exceed 4 (3 with E-710 3-axis versions). The assignment is valid until the E-710 is powered down or E7XX_WMS is called again.

Waveforms can be created based on predefined "curve" shapes. Additionally you can freely define curve shapes. The waveform can be made up by concatenating a number of “segments”. See the E7XX_WAV functions (p. 42 ff.) for more information. The waveforms are stored in the wave tables until the E-710 is powered down or the number of points per table is reset. Even if E7XX_WMS is called for only one wave table, the waveforms should be defined again for all tables.

A waveform can be output a fixed number of times, or repeated indefinitely (see E7XX_WGO, p. 45). When the wave generator output is stopped and restarted, it will continue with the first point of the waveform. The output values will always be interpreted as positions, so that the servo must be on during wave generator operation. Note that these target values are relative positions, i.e. they are added to any other current target contributions coming from move commands and / or from the analog input (for details see E-710 User Manual). To address a wave generator, you have to use the ID of the axis for which the generator shall be started (e.g. if you want to start wave generator output for axis Z, the wave generator ID must be Z). Wave generator output will continue even if the terminal or the program from which it was started is quit.

Dynamic Digital Linearization (DDL) is standard on 6-axis E-710s and available as an option on 3- and 4-axis units. It can be used to reduce residual tracking error in dynamic applications, i.e. while the wave generators run. Using DDL involves gathering data in tables in the controller during an initialization phase and then applying that data during subsequent wave generator operation. See E7XX_WGO (p. 45), E7XX_DDL (p. 38), E7XX_qDDL (p. 39) and E7XX_DTC (p. 39).

Each time a wave generator is started, data recording starts automatically for the corresponding axis (read the data with E7XX_qDRR, p. 23). The data for the individual axes is written to separate record tables. The record configuration can be done with E7XX_DRC (p. 19). Recording ends when the record table content has reached the maximum number of points (8192 per table). Recording can be restarted with E7XX_WGR (p. 45).

5.2.2. How to Use the Wave Generator

NOTES

Be sure that you have set correct waveform sequence before enabling wave output to avoid unpredictable stage response, such as overflow and vibration.

Using the wave generators is as follows:

1. Set the maximum number of wave points for the wave tables with E7XX_WMS.
2. Define the waveform using the E7XX_WAV functions (if necessary create the waveform by concatenating multiple segments).
3. Optionally: Check the waveform:
   After you sent the waveform definition to the wave table, it is always a good idea to check it by reading back the waveform sequence from the E-710 before actually outputting it. This can be done with E7XX_qGWD (p. 40).
4. Optionally: If you want to change the table rate for wave generator and data recording, set parameter 0x13000109 using E7XX_SPA. This parameter is available in command level 1 (see E7XX_CCL) and can be set in RAM only (not in EEPROM).
5. Optionally: If you want to output trigger signals during the wave generator output, configure the trigger lines with E7XX_TWS (p. 41).
6. Set servo on with E7XX_SVO (p. 33) for the axes for which you want to start the wave generator output. Wave generator output is only possible in closed-loop operation because the wave points are interpreted as target positions (but not as voltages as it would be required for open-loop operation).
7. Start the wave generator output and hence the motion of the axis with E7XX_WGO. Different start modes can be set with E7XX_WGO separately for each wave generator (= axis), for example, DDL initialization or DDL usage during the wave generator output.
   When starting the wave generator, recording is started automatically, and the data can be read with E7XX_qDRR (p. 23).
8. Optionally: Restart recording with E7XX_WGR.
9. Stop the wave generator output with E7XX_WGO.

You can check the wave generator activation status with E7XX_IsGeneratorRunning (p. 39).

5.2.3. Examples: Wave Generation and Output

The following examples show the wave generator usage based on GCS commands (can be sent with E7XX_GcsCommandset or using the command entry facilities of *NanoCapture*™ or *PZTControl*). The corresponding DLL functions can be used accordingly.

General:

In this example, axis 1 was renamed to X (see also the example in Section 5.1.1 on p. 14) to illustrate the interrelation between axis ID and wave generator ID / wave table ID.

Command	Comment
WMS X 3000	Set the maximum number of points for the wave table belonging to axis X to 3000
WAV X ….	Define a waveform for wave table X (axis X)
SVO X 1	Servo is switched ON for axis X
WGO X 1	Start output of wave generator X (axis X) immediately (synchronized by interrupt)
WGO X 0	Stop output of wave generator X (axis X)

Waveform examples:

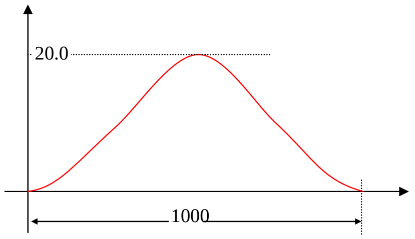
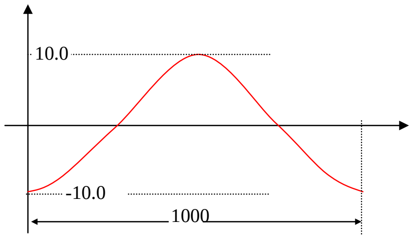
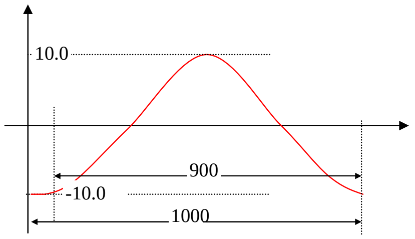
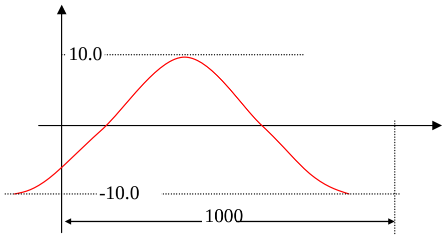
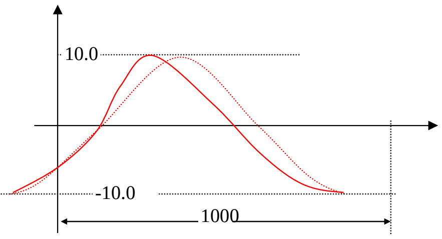
The examples below are all construed for axis 1, and the table rate is 1 (default).

NOTES

The table rate can be changed via the parameter 0x13000109 (command level 1; only in RAM)—this parameter sets the number of servo-loop cycles to be used for wave generator and data recording operations. Settings other than 1 make it possible to cover longer time periods with a limited number of points. By default, the servo update time is 200 µs, i.e. 5000 points = 1 s.

The offset in the waveform (*dOffsetOfWave* set by E7XX_WAV, p. 42) is ignored in the following cases:

- for the first segment of a waveform which consists of multiple segments
- when a waveform replaces the previous wave table content (*iAddAppendWave* = 0).

Description	Commands	Wave Form
Sine wave: 5 Hz, Amp$_{p-p}$= 20 µm	WAV 1 SIN_P 1000 20.0 WGO 1 1	
Sine wave: 5 Hz, Amp$_{p-p}$ = 20 µm Offset = -10 µm (*dOffsetOfWave)*	WAV 1 SIN_P 1000 20.0 -10.0 WGO 1 1	
Sine wave: 5 Hz, Amp$_{p-p}$ = 20 µm Offset = -10 µm (*dOffsetOfWave)*, 100 zeros (1-100)	WAV 1 SIN_P 1000 20.0 -10.0 900 WGO 1 1	
Sine wave: 5 Hz, Amp$_{p-p}$ = 20 µm Offset = -10 µm (*dOffsetOfWave)*, Phase shift 72°	WAV 1 SIN_P 1000 20.0 -10.0 1000 -200 WGO 1 1	
Sine wave: 5 Hz, Amp$_{p-p}$ = 20 µm Offset = -10 µm (*dOffsetOfWave)*, Phase shift 72° Type: asymmetrical	WAV 1 SIN_P 1000 20.0 -10.0 1000 -200 100 WGO 1 1	

5.2.4. Functions

- BOOL **E7XX_DDL** (int *ID*, int *iDdlTableId*, int *iOffsetOfFirstPointInDdlTable*, int *iNumberOfValues*, const double* *pdValueArray*)
- BOOL **E7XX_DTC** (int *ID*, int *iDdlTableId*)
- BOOL **E7XX_IsGeneratorRunning** (int *ID*, const char* *szWaveGeneratorIDs*, BOOL* *pbValueArray*)
- BOOL **E7XX_qDDL** (int ID, int *iDdlTableId*, int *iOffsetOfFirstPointInDdlTable*, int *iNumberOfValue*, double* *pdValueArray*)
- 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_WCL** (int *ID*, const long *iWaveTableId*)
- BOOL **E7XX_qWGO** (int *ID*, const char**szWaveGeneratorIds*, int* *iStartModArray*)
- BOOL **E7XX_qWMS** (int *ID*, const char* *szWaveTableIds*, int* *piMaxWaveSize*)
- BOOL **E7XX_TWC** (int *ID*)
- BOOL **E7XX_TWS** (int *ID*, const int* *piWavePointNumberArray*, const int* *piTriggerLevelArray*, long *iNumberOfPoints*)
- 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_WGO** (int *ID*, const char* *szWaveGeneratorIds*, const int* *iStartModArray*)
- BOOL **E7XX_WGR** (int *ID*)
- BOOL **E7XX_WMS** (int *ID*, const char* *szWaveTableIds*, const int* *iMaxWaveSize*)

5.2.5. Function Documentation

BOOL **E7XX_DDL** (int *ID*, int *iDdlTableId*, int *iOffsetOfFirstPointInDdlTable*, int *iNumberOfValues*, const double* *pdValueArray*)

Corresponding **command:** DDL

Transfer dynamic digital linearization feature data to a DDL data table on the E-7xx controller.

Notes:

Only the defined points in the selected DDL table on the controller are overwritten.

By default DDL table 1 belongs to axis 1, DDL table 2 to axis 2, …, up to DDL table 6 to axis 6. The assignment of axes to DDL tables can be changed using **E7XX_SPA**.

The data is stored on the controller only until a new DDL initialization is done with **E7XX_WGO** or the controller is powered down.

Arguments:

ID ID of controller

iDdlTableId number of the DDL data table to use. Table number can be 1 to 8.

iOffsetOfFirstPointInDdlTable index of first value to be transferred, (the first value in the DDL table has index 0)

iNumberOfValues number of values to be transferred

pdValueArray Array with the values for the DDL table (can have been filled with **E7XX_qDDL**).

Returns:

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

BOOL **E7XX_DTC** (int *ID*, const long *iDdlTableId*)

Corresponding **command:** `DTC`

Clears the linearization data of the dynamic digital linearization table (DDL feature required).

Arguments:

ID ID of controller

iDdlTableId variable with the ID of the data table which is to be cleared. Table number can be 1 to 8.

Returns:

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

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_qDDL** (int ID, int *iDdlTableId*, int *iOffsetOfFirstPointInDdlTable*, int *iNumberOfValue*, double* *pdValueArray*)

Corresponding **command:** `DDL?`

Get the dynamic digital linearization feature data from a DDL data table from the E-7xx controller. For large N values, communication timeout must be set long enough, otherwise a communication error may occur.

Notes:

By default DDL table 1 belongs to axis 1, DDL table 2 to axis 2, …, up to DDL table 6 to axis 6. The assignment of axes to DDL tables can be changed using **E7XX_SPA**.

The DDL data which is recorded if you select the Record DDL data option with **E7XX_DRC** can not be read with **E7XX_qDDL** but only with **E7XX_qDRR** and can therefore not be sent back to the controller with **E7XX_DDL**.

Arguments:

ID ID of controller

iDdlTableId number of the DDL data table. Table number can be 1 to 8.

iOffsetOfFirstPointInDdlTable index in the DDL table of first value to be read, the first value in the DDL table has index 0

iNumberOfValues number of values to be read

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

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. 40).

Every single point of the waveform must be sent over the interface so this can take several seconds, depending on the number of points.

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_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_TWC** (int *ID*)

Corresponding **command:** `TWC`

Trigger wave clear. Clears all triggers for the wave generators.

Arguments:

ID ID of controller

Returns:

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

BOOL **E7XX_TWS** (int *ID*, const int* *piWavePointNumberArray*, const int* *piTriggerLevelArray*, long *iNumberOfPoints*)

Corresponding **command:** `TWS`

Sets output trigger values for point(s) on the waveform. The corresponding values in the array *iTrigger* are bit-mapped. Each bit has the following meaning:

bit 0	trigger line 1: 0 not active, 1 active.
bit 1	trigger line 2: 0 not active, 1 active.
bit 2	trigger line 3: 0 not active, 1 active.
bit 3	trigger line 4: 0 not active, 1 active.
bit 8	if = 0, then the trigger values apply to corresponding *iWavePoint* point only if = 1, then the trigger values apply to all points between the last point set by this command and the corresponding *iWavePoint* point

Note:

The trigger values can be defined for a maximum number of 16,000 wave points.

Arguments:

ID ID of controller

piWavePointNumberArray array with the wave points.
piTriggerLevelArray array of bit-mapped integers specifying trigger values and their application.
iNumberOfPoints the number of points in the arrays *iWavePoint* and *iTrigger*.

Returns:

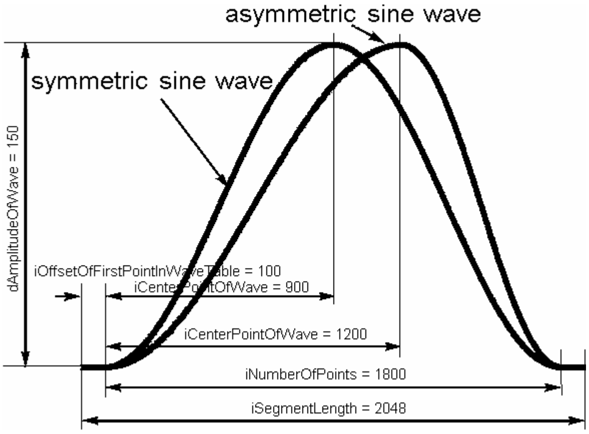
TRUE if no error, FALSE otherwise (see p. 7)

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. The wavepoint storage available for all wave tables together is 63488 points (62464 for 6-axis versions). The maximum number of points for each wave table can be set with **E7XX_WMS** (p. 46**)**. The offset *dOffsetOfWave* in the waveform is ignored in the following cases:

- for the first segment of a waveform which consists of multiple segments
- when a waveform replaces the previous wave table content (*iAddAppendWave* = 0).



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*)

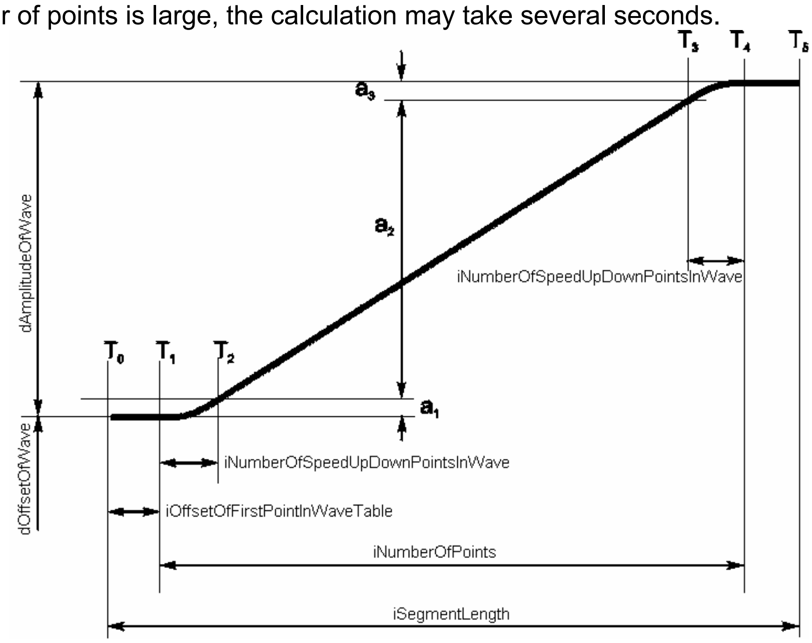
Corresponding **command:** `WAV LIN`

Have the wave generator produce a single line. The wave is not output at this time. The wavepoint storage available for all wave tables together is 63488 points (62464 for 6-axis versions). The maximum number of points for each wave table can be set with **E7XX_WMS** (p. 46**)**. The offset *dOffsetOfWave* in the waveform is ignored in the following cases:

- for the first segment of a waveform which consists of multiple segments
- when a waveform replaces the previous wave table content (*iAddAppendWave* = 0).

Note:

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


T3
T4
T5
a3
a2
a1
dAmplitudeOfWave
dOffsetOfWave
T0
T1
T2
iNumberOfSpeedUpDownPointsInWave
iNumberOfSpeedUpDownPointsInWave
iOffsetOfFirstPointInWaveTable
iNumberOfPoints
iSegmentLength

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. The wavepoint storage available for all wave tables together is 63488 points (62464 for 6-axis versions). The maximum number of points for each wave table can be set with **E7XX_WMS** (p. 46**)**.

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
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*)

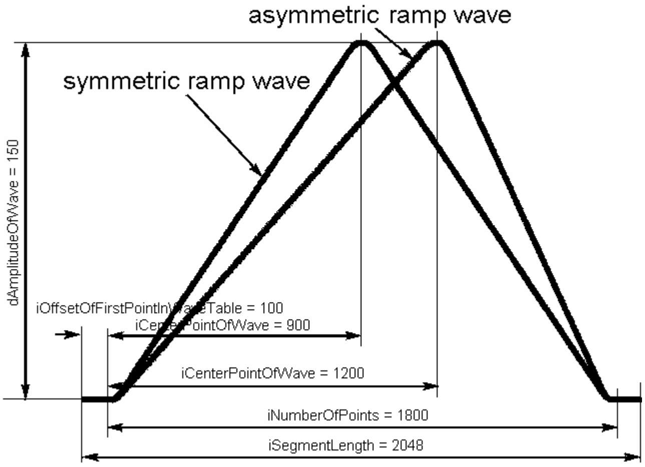
Corresponding **command:** `WAV RAMP`

Have the wave generator produce a ramp wave. The wave is not output at this time. The wavepoint storage available for all wave tables together is 63488 points (62464 for 6-axis versions). The maximum number of points for each wave table can be set with **E7XX_WMS** (p. 46**)**. The offset *dOffsetOfWave* in the waveform is ignored in the following cases:

- for the first segment of a waveform which consists of multiple segments
- when a waveform replaces the previous wave table content (*iAddAppendWave* = 0).

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.

Does also clear DDL 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_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_DRR**, p. 23. Recording can be restarted with **E7XX_WGR** (p. 45).

Two wave generators can run simultaneously. Recording is configured with **E7XX_DRC** (p. 19). When the wave generator is started by an external trigger signal (*iStartMod* = bit 1) and, in addition, the number of output cycles is limited (either by setting a certain value for parameter 0x13000003 or when *iStartMod* = bit 9 is used), E7XX_WGO must be called with *iStartMod* = 0 afterwards. Otherwise the E7XX_GCS_DLL could not recognize the current state of the wave generator.

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 is on the Digital I/O connector. There is one line for generator start and a second line for generator hold and start. See the E-710 User Manual for Digital I/O connector pinout.

bit 6: the dynamic digital linearization feature is used and reinitialized.

bit 7: the dynamic digital linearization feature is used.

bit 8: wave generator started at the endpoint of the last cycle.

bit 9: single run DDL test. Each following E7XX_WGR starts a new wave cycle and records the data specified with E7XX_DRC.

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** (p. 23).

The data type set with **E7XX_DRC** (p. 19) for record table 1 is recorded for all axis available on the controller.

Arguments:

ID ID of controller

Returns:

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

BOOL **E7XX_WMS** (int *ID*, const char* *szWaveTableIds*, int* *iMaxWaveSize*)

Corresponding **command:** `WMS`

Sets the maximum size of the wave storage for *szWaveTableIds*. A maximum number of 63488 points (62464 for 6-axis versions) can be defined for all wave tables together.

Note:

Due to memory restrictions, the number of tables to which the points can be distributed must not exceed 4 (3 with E-710 3-axis versions).

Arguments:

ID ID of controller
szWaveTableIds string with wave tables
iMaxWaveSize maximum size of the wave storage.

Returns:

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

5.3. GCS Command Syntax

To define the GCS syntax the following notation is used:

<name>	means	a category of the type name is used here.
[<parameter>]	means	<parameter> is optional. Between two parameters an optional space is always allowed.
{ <axis> }	means	repetition of <axis>.
text	means	text but if a single character is used it is marked with quotes like "A".
|	means	or
LF	means	Linefeed
CR	means	Carriage return
::=	means	"is defined as".
SP	means	a space character.

A GCS command consists of 3 characters, e.g. ABC. To the corresponding question command a "?" is appended e.g. ABC?. Additionally there are fast polling commands which consist only of one character. The 24th ASCII character e.g. is called #24. An exception is *IDN? to support GPIB. Historically there are four character commands which are still supported – but there will be no new ones.

Command mnemonic:

<CMD> ::= <character1><character2><character3>[?]

GCS syntax:

<CMD> SP { [<parameter>] } LF

Reply syntax:

[<parameter>"="] <value> LF

Multiple line reply syntax:

{ [<parameter>"="] <value> SP LF }

[<parameter>"="] <value> LF for the last line!

An <axis> is always identified with only one character.

Definitions for questioning commands ABC? { [<parameter>] } :

When no <parameter> is given it means all possible <parameters> should be replied. POS? asks for all <axis>, DIO? asks for all <relays>.

The command ABC? <par3><par1><par2> replies in the same order:

<par3>"="<value3> SP LF

<par1>"="<value1> SP LF

<par2>"="<value2> LF

The syntax is not case-sensitive.

6. System Parameter Overview

To adapt the E-710 to your application, you can modify parameter values—either for the whole system, for the axis, for the individual sensor and piezo channels or for the wave generators (see "Item Type" column in the table below; for the interdependence between axis and channels see the E-710 User Manual). Note that a parameter might have different values for each corresponding item, e.g. for each of the sensor channels. For parameters which refer to the whole system, *szAxes* is always 1.

The parameters depend on the controller firmware. With E7XX_qHPA you can obtain a list of all available parameters with detailed information. Note that many parameters are "protected" and can not be changed—it is only possible to change level-1-parameters (see E7XX_CCL for how to change the current command level).

Parameters can be changed temporarily or in non-volatile memory using the appropriate GCS commands in the command terminal (see E7XX_SPA, E7XX_SEP, E7XX_WPA).

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-710 controller calibrated at the factory for easy restoration of original settings after shipping. Some of the parameters are also saved in the ID chips which are housed in the connectors of the stages, see "Notes" column in the table below.

EEPROM and RAM copies of channel-1 and axis-1 parameter values are stored in memory bank 1, channel-2 and axis-2 values in memory bank 2, etc. See the description of the DP command in the User Manual for instructions on addressing the memory banks.

The following parameter numbers are valid in command level 1 (see **E7XX_CCL** on p. 17; parameter numbers for command level 0 are included):

Parameter Number	Item Type	Parameter Name	Range	Notes
0X02000000	Sensor Channel	Sensor Mechanic: Sensor/Analog enable	0 = Disabled 1 = Enabled	
0X02000100	Sensor Channel	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 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	ID-Chip

Parameter Number	Item Type	Parameter Name	Range	Notes
			26 = Option 0.68X 76 27 = Board Range 0.56X	
0X02000101	Sensor Channel	Sensor Mechanic: Board Gain	0 = Gain 0.5 64 = Gain 1.0 128 = Gain 2.0 192 = Gain 3.0	ID-Chip
0X02000102	Sensor Channel	Sensor Mechanic: Electrical poti selected		ID-Chip
0X02000200	Sensor Channel	Sensor Mechanic: Sensor correction 0 order		
0X02000300	Sensor Channel	Sensor Mechanic: Sensor correction 1st order		
0X02000400	Sensor Channel	Sensor Mechanic: Sensor correction 2nd order		
0X02000500	Sensor Channel	Sensor Mechanic: Sensor correction 3rd order		
0X02000600	Sensor Channel	Sensor Mechanic: Sensor correction 4th order		
0X05000000	Sensor Channel	Sensor Filter: Digital filter type	0 = No Filter 1 = IIR Filter 2 = FIR filter	
0X05000001	Sensor Channel	Sensor Filter: Digital filter Bandwidth/Hz		
0X05000002	Sensor Channel	Sensor Filter: Digital filter order		
0X05000101	Sensor Channel	Sensor Filter: User filter parameter A0		
0X05000102	Sensor Channel	Sensor Filter: User filter parameter A1		
0X05000103	Sensor Channel	Sensor Filter: User filter parameter B0		
0X05000104	Sensor Channel	Sensor Filter: User filter parameter B1		
0X05000105	Sensor Channel	Sensor Filter: User filter parameter B2		
0X07000000	Axis	Servo: Range min limit (µ)		ID-Chip
0X07000001	Axis	Servo: Range max limit (µ)		ID-Chip
0X07000200	Axis	Servo: Servo loop slew rate (axis unit/ms)		ID-Chip
0X07000300	Axis	Servo: Servo loop P-Term		ID-Chip
0X07000301	Axis	Servo: Servo loop I-Term		ID-Chip
0X07000500	Axis	Servo: Position from sensor 1		ID-Chip
0X07000501	Axis	Servo: Position from sensor 2		ID-Chip
0X07000502	Axis	Servo: Position from sensor 3		ID-Chip

Parameter Number	Item Type	Parameter Name	Range	Notes
0X07000503	Axis	Servo: Position from sensor 4		ID-Chip
0x07000504	Axis	Servo: Position from sensor 5		ID-Chip
0x07000505	Axis	Servo: Position from sensor 6		ID-Chip
0x07000506	Axis	Servo: Position from sensor 7		ID-Chip
0x07000507	Axis	Servo: Position from sensor 8		ID-Chip
0X07000800	Axis	Servo: sensor ON/OFF start up	0 = Disabled 1 = Enabled	
0X07000801	Axis	Servo: Servo enable	0 = Disabled 1 = Enabled	
0X07000802	Axis	Servo: Auto-Zero start up	0 = Disabled 1 = Enabled	
0X07000900	Axis	Servo: Tolerance		ID-Chip
0X07000A00	Axis	Servo: Auto-Zero driving low voltage (V)		ID-Chip
0X07000A01	Axis	Servo: Auto-Zero driving high voltage (V)		ID-Chip
0X07000B00	Axis	Servo: Zoom Auto-Zero low voltage (V)		ID-Chip
0X07000B01	Axis	Servo: Zoom Auto-Zero high voltage (V)		ID-Chip
0X07000C00	Axis	Servo: Default position		ID-Chip
0X07000C01	Axis	Servo: Default voltage		ID-Chip
0X07010100	Axis	Servo: Syncronous		
0X08000100	Axis	Servo output filter: Notch frequency of filter nr. 1		
0X08000101	Axis	Servo output filter: Notch frequency of filter nr. 2		
0X08000200	Axis	Servo output filter: Notch rejection of filter nr. 1		
0X08000201	Axis	Servo output filter: Notch rejection of filter nr. 2		
0X08000300	Axis	Servo output filter: Notch bandwidth of filter nr. 1		
0X08000301	Axis	Servo output filter: Notch bandwidth of filter nr. 2		
0X08000400	Axis	Servo output filter: Creep factor T1 (sec)		ID-Chip
0X08000401	Axis	Servo output filter: Creep factor T2 (sec)		ID-Chip
0X09000000	Axis	Output Matrix: Driving with piezo 1		
0X09000001	Axis	Output Matrix: Driving with piezo 2		
0X09000002	Axis	Output Matrix: Driving with piezo 3		
0X09000003	Axis	Output Matrix: Driving with piezo 4		

Parameter Number	Item Type	Parameter Name	Range	Notes
0x09000004	Axis	Output Matrix: Driving with piezo 5		
0x09000005	Axis	Output Matrix: Driving with piezo 6		
0x09000006	Axis	Output Matrix: Driving with piezo 7		
0x09000007	Axis	Output Matrix: Driving with piezo 8		
0X0C000000	Piezo Channel	Piezo: Output voltage low limit (V)		ID-Chip
0X0C000001	Piezo Channel	Piezo: Output voltage high limit (V)		ID-Chip
0X0E000100	System	System Global: Sensor sampling time		
0X0E000200	System	System Global: Servo update time		
0X0E000300	System	System Global: Trigger-in enable (D_IN/Bit)		
0X0E000400	System	System Global: DDL-License		
0X0E000500	System	System Global: Last firmware version		
0X0E000800	System	System Global: Trigger active high/low	-1 = Active Low 1 = Active High	
0X0E000900	System	System Global: Trigger pulse length	1 = Short Pulse 2 = Long Pulse	
0X0F000000	Axis	System Mechanic: Power Up Read ID-Chip Enable		
0X0F000100	Axis	System Mechanic: Stage type		
0X0F000200	Axis	System Mechanic: Serial number		ID-Chip
0X10000000	Axis	Fast Interface: DSP-Link input low limit		
0X10000001	Axis	Fast Interface: DSP-Link input high limit		
0X10000200	Axis	Fast Interface: PIO input low limit		
0X10000201	Axis	Fast Interface: PIO input high limit		
0X10000300	System	Fast Interface: Interface PIO control	0 = Disabled 1 = Enabled	
0X10000303	System	Fast Interface: PIO on target line 4 function	0 = Trigger 8 = Axis 4 On-Target signal	
0X12000000	System	Interface Parser: GPIB address		
0X12000100	System	Interface Parser: Serial port default baud rate		
0X13000000	Wave Generator	Wave Generator: Generator running control	0 = All Disabled 16 = Single Run Enabled 48 = SYNC Run Enabled 80 = Run/DDL Enabled 240 = All Enabled	
0X13000001	Wave Generator	Wave Generator: Installed wave form		

Parameter Number	Item Type	Parameter Name	Range	Notes
0X13000002	Wave Generator	Wave Generator: Connected axis		
0X13000003	Wave Generator	Wave Generator: Wave generator cycles		
0X13000101	Wave Generator	Wave Generator: Curve shape	0 = No Function 16 = Sine Wave 32 = Ramp Wave 64 = Single Line	
0X13000102	Wave Generator	Wave Generator: Total wave form points		
0X13000103	Wave Generator	Wave Generator: Curve points		
0X13000104	Wave Generator	Wave Generator: Center point		
0X13000105	Wave Generator	Wave Generator: Speed-up/Slow-down points		
0X13000106	Wave Generator	Wave Generator: Curve start point		
0X13000107	Wave Generator	Wave Generator: Curve amplitude		
0X13000108	Wave Generator	Wave Generator: Curve offset		
0X13000109	System	Wave Generator: Wave generator table rate		In RAM only, not in EEPROM[1]
0X14000000	Axis	DDL: DDL table number		
0X14000001	Axis	DDL: DDL repeat number		
0X14000002	Axis	DDL: DDL gain constant		
0X14000003	Axis	DDL: DDL gain change		
0X14000004	Axis	DDL: DDL gain curve		
0X14000005	Axis	DDL: Final DDL Gain		
0X14000006	Axis	DDL: Final delay max (s)		
0X14000007	Axis	DDL: Final delay min (s)		
0X14000008	Axis	DDL: Time delay change rule		
0X14000009	Axis	DDL: Final time delay		
0X1400000A	Axis	DDL: DDL zero gain number		
0X14000200	Axis	DDL: DDL report filter	0 = Disabled 1 = Enabled	
0X14000201	Axis	DDL: DDL report filter band width factor		

[1] This parameter sets the number of servo-loop cycles to be used for wave generator and also for data recording operations.

7. 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 wav 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	GCS-array doesn't support different length; request arrays which have different length 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	MOV, MVR, SVA, SVR, STE, IMP and WGO blocked when analog target is active
73	PI_CNTR_WAVE_GENERATOR_ACTIVE	MOV, MVR, SVA, SVR, STE and IMP blocked when wave generator is active
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	Position can be estimated only while MOV! is running
219	PI_CNTR_POSITION_BASED_ON_INTERPOLATION	Position was calculated while MOV is running
301	PI_CNTR_SEND_BUFFER_OVERFLOW	Send buffer overflow
302	PI_CNTR_VOLTAGE_OUT_OF_LIMITS	Voltage out of limits
303	PI_CNTR_VOLTAGE_SET_WHEN_SERVO_ON	Attempt to set voltage 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
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
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 non-blocked signal

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

8. Index