Piezo · Nano · Positioning

MS154E Software Manual

Mercury™ Class

PI_Mercury_GCS_DLL

Release: 1.0.1 Date: 2007-12-19

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

- C-863
  Mercury™ Networkable Single-Axis DC-Motor Controller
- C-862
  Mercury™ Networkable Single-Axis DC-Motor Controller
- C-663
  Mercury™ Step Networkable Single-Axis Stepper Motor Controller

Moving the NanoWorld | www.pi.ws

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

Physik Instrumente (PI) GmbH & Co. KG is the owner of the following company names and trademarks:
PI®, PIMikroMove®, Mercury™, Mercury Step™
The following designations are protected company names or registered trademarks of third parties:
Windows, LabView

First printing 2007-12-19
Document Number MS154E, Release 1.0.1
PI_Mercury_GCS_DLL_MS154E.doc

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 This manual is superseded by any new release.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

About This Document

Users of This Manual

This manual assumes that the reader has a fundamental understanding of basic servo systems, as well as motion control concepts and applicable safety procedures.
The manual describes the PI General Command Set (GCS) Windows DLL for Mercury™ Class controllers. With present firmware, all software which accepts GCS commands must pass them to the controller via this DLL or the corresponding COM Server.
This document is available as PDF file on the product CD. For updated releases see www.pi.ws, contact your PI Sales Engineer or write info@pi.ws.

Conventions

The notes and symbols used in this manual have the following meanings:

CAUTION

Calls attention to a procedure, practice, or condition which, if not correctly performed or adhered to, could result in damage to equipment.

NOTE

Provides additional information or application hints.

Related Documents

The Mercury™ controller and the software tools which might be delivered with the controller are described in their own manuals (see below). All documents are available as PDF files via download from the PI Website (www.pi.ws) or on the product CD. For updated releases contact your Physik Instrumente Sales Engineer or write info@pi.ws.

Hardware User Manuals	User Manuals for all hardware components
Mercury GCSLabVIEW_MS149E	LabView VIs based on PI GCS command set
Mercury GCS DLL_MS154E	WindowsGCS-based DLL Library (this document)
PIMikroMove User Manual SM148E	PIMikroMove® Operating Software (GCS-based)
Mercury Commands MS163E	Mercury™ GCS Commands
PIStageEditor _SM144E	Software for managing GCS stage-data database
MMCRun MS139E	Mercury Operating Software (native commands)
Mercury Native DLL & LabVIEW MS177E	Windows DLL Library and LabView VIs (native-command-based)
Mercury Native Commands MS176E	Native Mercury™ Commands

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 MERCURY™ GCS DLL

The PI_Mercury_Class_GCS_DLL allows controlling one or more PI Mercury™ Class controller networks, each consisting of one or more Mercury™ Class controllers. Each network is connected to a host PC via a single RS-232 or USB port.

NOTE

Multiple controllers on a single host computer USB or RS-232 interface are interconnected using a RS-232 bus architecture. The host communicates with one Mercury™ Class device at a time. Such a network appears to the MERCURY™ GCS DLL user as a single, multi-axis controller and is usually referred to in this manual as a “controller network”.

1.1. Quick Start

1.1.1. Software Installation

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

- Be sure to login to the host PC with administrator rights
- If the Setup Wizard does not open automatically, start it from the root directory of the CD with the setup icon.
- Follow the on-screen instructions. You can choose between “typical” and “custom” installation. Typical components are GCS LabView drivers, Native and GCS DLLs, *PIMikroMove®, MMCRun* and all manuals. “Typical” is recommended.
- Sample programs and the appropriate source code are to be found in the \Sample directory of the product CD.

1.1.2. Connect the Controller

CAUTION !

Never connect the RS-232-IN and USB connectors of the same controller to a PC at the same time, as damage may result.

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

1.1.3. Install USB Drivers

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

NOTE

The USB drivers make the USB interface appear to the software as an additional RS-232 COM port. That port is present only when the Mercury™ USB device is connected and powered up.

To initiate communication, use the DLL functions described in "Communication Initialization" on p. 17.

1.2. General Command Set (GCS)

It is possible to use either the Mercury™ native ASCII command set or the PI General Command Set (GCS) to operate a Mercury™ class controller. .The native ASCII command set is understood by all versions of the controller firmware directly (see the Mercury Native Commands manual for details). GCS, the PI standard command set, offers compatibility between different controllers. With current firmware, GCS command support is implemented by the Windows DLL described in this manual which translates the GCS commands to the native commands. Once the PI Mercury Class_GCS_DLL.dll library is installed, you can use, for example, the LabVIEW GCS drivers to control a Mercury™ class 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.

NOTE

Although the GCS DLL has a gateway for sending native commands, mixing native and GCS commands is not recommended. GCS move commands, for example, may not work properly after the position has been changed by a native command.

1.3. Axes and Stages

Mercury™ Class controllers can be chained together on an RS-232 bus network and all controlled through one port of the host computer (USB or RS-232). One that network, native commands are used, and the commands and responses are always sent between the host computer and one *selected* controller, with the other controllers in the *deselected* state.

The GCS DLL makes a network of Mercury™ Class controllers connected to one port look like one controller with up to 16 axes (if host’s RS-232 port is used, number of usable axes may be limited to as few as 6 by current available). See the controller User Manual for information on setting the controller device number (1 to 16); typically 4 address DIP switches are used. The device number determines the default identifiers of the corresponding axes and I/O channels.

1.3.1. Axis Designators

By default the axes are named “A” to “P”. The axis connected to the Mercury™ controller with device number 1 will be addressed as axis “A” in the GCS DLL, the Mercury™ No. 5 will provide axis “E”, etc. If these two controllers are the only ones connected, the GCS DLL will provide only the two axes “A” and “E”.

1.3.2. I/O Line Designators

Each Mercury™ and Mercury™ Step controller provides four analog/digital input and four digital output lines. For digital IO, these channels are named with the characters

```
ABCD   EFGH   IJKL   MNOP   QRST   UVWX   YZ12   3456   7890   @?>=   <;:`
       _^]\   [/.-   ,+*)   ('&%   $#"!
```

in groups of 4, one group for each of the 16 possible controller addresses.

For analog input (0 to 5 V), the input channels of a Mercury™ Class network have IDs from 1 to 64—again 4 x 1 less than the device number is added to the line number to give the channel number. Note that for C-862 controllers, the last channel is digital-only.

Example: A network with a C-862 DC Motor Controller with device number 1 and a C-663 Stepper Controller with device number 3. The GCS DLL will provide

- Axes "A" and "D"
- Digital I/O using channel IDs A, B, C, D, M, N, O and P
- Analog input using channel IDs 1-3 and 25-28

1.3.3. Controller Joystick Connections

Each axis associated with a controller having a joystick port, can be associated with one axis of motion of a joystick. That axis, and the associated joystick button, is identified in the network by the controller device number. Note that the included joystick Y-cable permits connecting one axis and one logical button of one joystick to one controller and the other axis and other button to another controller.

1.4. 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.

1.5. Overview

This document describes the general handling of GCS DLLs and the individual functions of the MERCURY GCS DLL. You can also use this document when you are working with the GCS COM server—see Section 6 on p. 12 for the COM server special features.

- Units and GCS (p. 9) explains the units used for commanding positions
- Referencing (p. 9) explains how to properly initialize your system and the connected stages
- DLL Handling (p. 9) explains how to load the library and how to access the functions provided by the MERCURY DLL.
- Function Calls (p. 11) and Types Used in PI Software (p. 11) provides some general information about the syntax of most commands in the DLL.
- GCS COM Server (p. 12) points out the differences between DLL and COM server handling.
- Native Command Gateway (p. 15) shows how to initiate communication with a Mercury™ Class controller or controller network (see also Interface Settings (p. 20)).
- Mercury™ Class Commands (p. 20) describes the functions encapsulating the embedded GCS commands for Mercury™ Class controllers
- Motion Parameters Overview (p. 42) describes how to handle the stage parameters and list the valid parameter set.
- Error Code (p. 50) has a description of the possible errors.

1.6. Units and GCS

1.6.1. Hardware, Physical Units and Scaling

The GCS (General Command Set) system uses basic physical units of measure. Most controllers and GCS software have default conversion factors chosen to convert hardware-dependent units (e.g. encoder counts) into millimeters or degrees, as appropriate (see Mercury_SPA and Mercury_qSPA descriptions, parameters 14 and 15). The defaults are generally taken from a database of stages that can be connected. An additional scale factor can be applied (see Mercury_DFF), to the basic physical unit making a working physical unit available without overwriting the conversion factor for the first. This is the unit referred to by the term "physical unit" in the rest of this manual.

1.6.2. Rounding Considerations

When converting move commands in 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 physical 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 physical 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.

2. Referencing

Upon startup (or after a call to **Mercury_INI** ()) a controller has no way of knowing the absolute position of a connected axis. The axis is said to be “unreferenced” and no moves can be made. Moves can be made allowable in the following ways:

- The axis can be referenced. This involves moving it until it trips a reference or limit switch. See the Mercury_REF, Mercury_MNL and Mercury_MPL functions for details
- The controller can be told to set the reference mode for the axis OFF and allow relative moves only, without knowledge of the absolute position. See the Mercury_RON function for details.
- For axes with reference mode OFF, the controller can be told to assume the absolute position has a given value. See the Mercury_POS function for details.

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

3.1. Using a Static Import Library

The `PI_Mercury_GCS_DLL.DLL` module is accompanied by the `PI_Mercury_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:

1. 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.

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

3.2. 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:

1. 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 need a `WINAPI` or `__stdcall` modifier in the declaration.
2. Modify the module definition file with an `IMPORTS` section. In this section, all functions used in the program must be named. Follow the syntax of the `IMPORTS` statement. Example:

```
IMPORTS
   PI_Mercury_GCS_DLL.Mercury_IsConnected
```

3.3. 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:

1. 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_Mercury_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_Mercury_IsConnected)( int );
   FP_Mercury_IsConnected p Mercury _IsConnected;
```

2. 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("PI_Mercury_GCS_DLL.DLL\0");
```

3. 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:

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

4. Function Calls

Almost all functions will return a boolean value of type BOOL (see “Types Used in PI Software” (p. 11)). If the function succeeded, the return value is **TRUE**, otherwise it is **FALSE**. To find out what went wrong, call **Mercury_GetError**()(p. 18)) and look up the value returned in “Error Code” (p. 50). The first argument to most function calls is the ID of the selected controller network.

4.1. Controller ID

The first argument to most function calls is the ID of the selected controller network. To allow the handling of multiple controller networks, the DLL returns a non-negative "ID" when a connection to a controller network is opened. This is a kind of index to an internal array storing the information for the different controller networks. All other calls addressing the same controller network require this ID as first argument. The individual Mercury™ Class controllers in a Mercury™ controller network are distinguished by the axes which they control.

4.2. Axis Identifiers

Many functions accept one or more axis identifiers. If no axes are specified (either by giving an empty string or a **NULL** pointer) some functions will address all connected axes. In a Mercury™ Class controller network, the different axes correspond to the different individual controllers.

4.3. Axis Parameters

The parameters for the 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 `Mercury_qPOS("ABC", double pos[3])`, the position for 'A' is in `pos[0]`, for 'B' in `pos[1]` and for 'C' in `pos[2]`.

Axes: `szAxes = "ABC"`	Positions:`pos = {1.0, 2.0, 3.0}`
`szAxes[0] = 'A'`	`pos[0] = 1.0`
`szAxes[1] = 'B'`	`pos[1] = 2.0`
`szAxes[2] = 'C'`	`pos[2] = 3.0`

If you call `Mercury_MOV("AC", double pos[2])` the target position for 'A' is in `pos[0]` and for 'C' in `pos[1]`.

Each axis identifier is sent only once. Only the **last** occurrence of an axis identifier is actually sent to the controller with its argument. Thus, if you call
`Mercury_MOV("AAB", pos[3])` with `pos[3] = { 1.0, 2.0, 3.0 }`, 'A' will move to 2.0 and 'B' to 3.0. If you then call `Mercury_qPOS("AAB", pos[3])`, `pos[0]` and `pos[1]` will contain 2.0 as the position of 'A'.

(See **Mercury_MOV**() (p. 30) and **Mercury_qPOS**() (p. 35))

See “Types Used in PI Software” (p. 11) for a description of types used for parameters.

5. Types Used in PI Software

5.1. 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
```

5.2. 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
```

5.3. 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`".

6. GCS COM Server

For some programming languages it is much simpler to use a COM server than to link to DLL functions. Mainly Visual Basic and other script languages (e.g. Python, Perl) provide good support for calling COM functions. See the provided samples for ways to integrate the GCS COM into the different languages / development environments. Sample programs and the appropriate source code are to be found in the \Samples directory of the product CD.

The functions are more or less the same as provided by the DLL, so this manual can be used to get to know the basic functionality. There are however fundamental syntax differences:

- No controller ID, since you can create instances of the COM object for every single controller network connected (see Section 6.1)
- With COM it is possible to allocate space for strings and arrays by the callee without disturbing the caller, so there is no need to send any buffer sizes or array lengths to the COM functions (see Section 6.2)
- It is possible to have "properties" which not only set values but also trigger certain functions (see Section 6.3)

6.1. No Need for Controller IDs

You can create instances for every controller network connected. Below is an example of equivalent C or C++ and Visual Basic code:

```
int ID1;
int ID2;
ID1 = Mercury ConnectRS232(1, 115200);
ID2 = Mercury ConnectRS232(2, 115200);

if (!Mercury IsConnected(ID1))
    printf("Could not connect to controller 1";
if (!Mercury IsConnected(ID2))
    printf("Could not connect to controller 2";
```

C or C++ code

```
Dim MERCURY1 As New MERCURY
```

```
Dim MERCURY2 As New MERCURY

MERCURY1.ConnectRS232(1, 115200)
MERCURY2.ConnectRS232(2, 115200)

If Not MERCURY1.IsConnected Then
    Me.Caption = "Could not connect to controller 1"
End If
If Not MERCURY2.IsConnected Then
    Me.Caption = "Could not connect to controller 2"
End If
```

Visual Basic code

6.2. No Need for Buffer Sizes

If you want to read a string with a DLL functions from the DLL, you need to allocate the neccessary space and tell the DLL how large the buffer is. The COM server, however, expects a “string object". The COM server can let the string grow and the string object itself holds all the neccessary information about length and memory requirements. Thus the following C or C++ and Visual Basic code is equivalent:

```
char sIDN[1024];
Mercury_qIDN( ID, sIDN, 1024 );
```

C or C++ code

```
Dim sIDN As String
MERCURY.qIDN( sIDN )
```

Visual Basic Code

6.3. COM Properties

A COM server can have so-called *properties*. These behave like ordinary variables, but if you read from or write to them, an internal function is triggered (not every property needs to support both reading and writing). Most GCS COM servers have a property "moving". So you do not need to call IsMoving() but can simply use (read) that property and a call to IsMoving() is generated internally. Some GCS COM servers have properties for many axis identifiers. If you assign a new value to one of these properties and the corresponding axis is connected, a MOV is sent. If you read from such a property, the COM will first call POS? and then set the value.

Here are two more blocks of equivalent code:

```
BOOL bIsReferencing;
do
{
   Sleep(100);
   Mercury IsReferencing(ID, "", &bIsReferencing);
} while (bReferencing == TRUE);

Mercury MOV(ID, "A", 10);
Sleep(1000);
double currentPos;
Mercury_qPOS(ID, "A", &currentPos);
```

C or C++ code

```
Do
    Sleep 100
```

```
Loop While MERCURY.Referencing ' wait until referenced

MERCURY.A = 10;
Sleep 1000;
Dim currentPos As Double
currentPos = MERCURY.A
```

Visual Basic Code

7. Native Command Gateway

The GCS DLL includes a function which provides access to all the commands of the controller's native command set. Use of this set is only recommended for users who have already worked with this command set and do not want to learn the GCS command set. The General Command Set should be preferred because of its compatibility with other PI controllers.

The GCS DLL function calls giving access to native commands/responses are as follows:

- BOOL **Mercury_ReceiveNonGCSString**(intID, char* szString, int iMaxSize);
- BOOL **Mercury_SendNonGCSString**(intID, const char* szString);

BOOL **Mercury_ReceiveNonGCSString** (int *ID*, char * *szAnswer*, int *bufsize*)

Gets the answer to a native command of one of the Mercury™s in the network, provided its length does not exceed *bufsize*. 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 Mercury Native Commands manual for a description of the native commands which are understood by the firmware, and for a command reference.

Arguments:

ID ID of controller
szAwnser the buffer to receive the answer.
bufsize the size of *szAnswer*.

Returns:

TRUE if no error, FALSE otherwise

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

Sends a native command to one of the Mercury™s in the network. 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.

If you want to address different controllers, the native-command, two-character address selection code can also be sent with this function (see the Mercury™ Native Commands manual for details)

```
char addr[3];
addr[0] = 1;
addr[1] = 'A'; // for mercury with address 0
addr[2] = '\0';
Mercury_SendNonGCSString(ID, addr);
```

See the Native Commands manual for a description of the native commands which are understood by the firmware, and for a command reference.

Arguments:

ID ID of controller
szCommand the GCS command as string.

Returns:

TRUE if no error, FALSE otherwise

8. Functions for User-Defined Stages

The PI Mercury GCS DLL also has functions allowing you to both define and save new stages (parameter sets).

Being able to specify the parameters of a stage and then save those parameters as a set under the stage name makes it easier to connect to previously defined stages. New (user-defined) stages are all stored in *MERCURYUserStages.dat* and known PI stages are in *PiStages.dat.* For parameter descriptions see the “Parameter List” Section (p. 43).

Two separate mechanisms are provided for the use of stage parameter sets:

- You can execute a function call that puts the *PIStageEditor* (a GUI dialog) on the screen where your user can set the stage parameters as he or she desires. See the separate PI Stage Editor manual for a description of how to operate that graphic interface.
- You can put the desired values in variables and execute function calls for setting the parameters and manipulating the parameter sets. See the function descriptions and the parameter ID list on p. 43 for details.

In either case, the procedure involves optionally loading a parameter set (connecting a stage) from the list of stage names in the *.dat* files, perhaps then deleting that stage (user-defined stages only) or editing the current, active parameters and saving them under a “new” name (to *MercuryUserStages.dat*) *.* It is not possible to edit *MercuryUserStages.dat* directly*:* all changes go via the currently active parameter set. *PiStages.dat* may not be edited at all, but updated versions should be made available regularly from PI.

8.1. Function Calls to Edit, Remove and Add Stage Definitions

Note that the parameter which determines whether a stage is “new” or not is the *Name* parameter. If there is no*Name* specified*,* the parameter set is not valid. Only when the current parameter set is valid can you, for example, call INI.

To create a valid parameter set for a new stage, you can use the Mercury_SPA function call (p. 41).

You can ease the creation by loading an existing parameter set with **CST** (p.22) and afterwards change the name and any other parameters, which differ, with SPA. (The **CST** command “connects” a valid stage, i.e. makes its parameter set active. It uses the corresponding parameters in the DAT files, so that you do not have to set them all by yourself.)

To save a new stage and thus make it available for a future connection with **CST,** use **Mercury_AddStage**() (p.) to add its parameter set to *MercuryUserStages.dat*. After addition to *MERCURYUserStages.dat* the stage will also appear in the list returned by **VST?** (p.).

If you want to remove a stage from *MercuryUserStages.dat* call *Mercury_***RemoveStage**() (p.17).

If you want to change parameters in *MercuryUserStages.dat* directly, call **Mercury_OpenUserStagesEditDialog**() to open it with the *PIStageEditor*. With **Mercury_OpenPiStagesEditDialog**() you can open the *PiStages.dat* with the *PIStageEditor*, but the file is protected and can not be changed. However with the *PIStageEditor* it is possible to save *PiStages.dat* under a new name (in the same directory) and edit this new file.

Notes:

The GCS DLL only accepts the DAT-files PiStages.dat and *Mercury*UserStages.dat. Although it is possible to save DAT-files with any user-defined names, they are not used by the software.

The **CST** (p.22*)* and **VST?** (p.39*)* commands look for the files *Mercury*UserStages.dat and PiStages.dat in the directory of the executable (EXE) file. If you have selected the *Typical* setup type, this directory is set automatically to *C:\<Program Files>\PI\GcsTranslator* (default). If you choose the *Custom* setup type, you can specify another directory. In that case the **CST** (p. 22*)* and **VST?** (p. 39*)* commands will look there for the *Mercury*UserStages.dat and PiStages.dat files.

8.2. Stage Definition Function Overview

```
BOOL Mercury_AddStage (const ID, const char* szAxes)
BOOL Mercury_RemoveStage (int ID, const char *szStageName)
```

BOOL **Mercury_OpenUserStagesEditDialog** (int ID)
BOOL **Mercury_OpenPiStagesEditDialog** (int ID)

8.3. Stage Parameter IDs

When defining user stages, it is important to set the stage parameters correctly. See the Mercury_qSPA function call on p. 36 for the parameters most frequently accessed by the user, for a complete list see the “Parameter List” Section (p. 43).

BOOL **Mercury_AddStage** (const int *iId*, char *const *szAxes*)

Adds the stage of the specified *axis* to the file *MercuryUserStages.dat* with the user defined stages.

Arguments:

iId ID of controller
szAxes character of the axis.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_RemoveStage** (const int *iId*, char * *szStageName*)

Removes the stage with the given name from the *MercuryUserStages.dat* file, which contains the user-defined stages.

Arguments:

iId ID of controller
szStageName the stage name as string.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_OpenPiStagesEditDialog** (const int *iId*)

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

Arguments:

iId ID of controller

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_OpenUserStagesEditDialog** (const int *iId*)

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

Arguments:

iId ID of controller

Returns:

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

9. Communication Initialization

9.1. Functions

- int **Mercury_ConnectRS232** (int nPortNr, long BaudRate)

- int **Mercury_InterfaceSetupDlg** (const char* szRegKeyName, BOOL bShowDetails)
- BOOL **Mercury_IsConnected** (int ID)
- void **Mercury_CloseConnection** (int ID)
- int **Mercury_GetError** (int ID)
- BOOL **Mercury_TranslateError** (int errNr, char *szBuffer, int maxlen)
- BOOL **Mercury_SetErrorCheck** (int ID, BOOL bErrorCheck)

9.2. Detailed Description

To use the DLL and communicate with a Mercury™ class controller or controller network, the DLL must be initialized with one of the "open" functions **Mercury_InterfaceSetupDlg**() or **Mercury_ConnectRS232**(). To allow the handling of multiple controller networks, the DLL will return a non-negative "ID" when one of these functions is called. This is a kind of index to an internal array storing the information for the different controller networks. All other calls addressing the same controller network have this ID as first parameter. **Mercury_CloseConnection**() will close the connection to the specified controller network and free its system resources.

9.3. Function Documentation

void **Mercury_CloseConnection** (int ID)

Close connection to Mercury Class controller network associated with *ID*. *ID* will not be valid any longer.

Arguments:

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

int **Mercury_ConnectRS232** (int *nPortNr*, long *BaudRate*)

Open an RS-232 ("COM") interface to a controller. All future calls to control this controller need the ID returned by this call.

Arguments:

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

Returns:

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

int **Mercury_GetError** (int ID)

Get error status; if there is no error set in the library, this function will call **Mercury_qERR**() (p. 32) to determine the error status in one of the controllers in the network. Any error returned is also cleared.

Returns:

error ID, see **Error codes** (p. 50) for the meaning of the codes.

int **Mercury_InterfaceSetupDlg** (const char* *szRegKeyName*)

Open dialog to let user select the interface and create a new Controller object. All future calls to control this Mercury™ Network need the ID returned by this call. See **Interface Settings** (p. 20) 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\Mercury_DLL"` is used.

Note:

If your programming language is C or C++, use "\\" to represent a single "\" in a literal: for example to create `"MyCompany\Mercury_DLL"` you must call

```
Mercury_InterfaceSetupDlg( "MyCompany\\Mercury_DLL" )
```

Returns:

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

BOOL **Mercury_IsConnected** (int ID)

Check if there is a Mercury™ Class controller network with an ID of *ID*.

Returns:

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

BOOL **Mercury_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 currently selected controller on the controller network (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 **Mercury_GetError**() (p. 18) 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 network
bErrorCheck switch error checking on (**TRUE**) or off (**FALSE**)

Returns:

the previous state, i.e before this call

BOOL **Mercury_TranslateError** (int *errNr*, char * *szBuffer*, int *maxlen*)

Translate error number to error message.

Arguments:

errNr number of error, as returned from **Mercury_GetError**()(p. 18).
szBuffer pointer to buffer that will store the message
maxlen size of the buffer

Returns:

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

9.4. Interface Settings

See the controller user manual for hardware connection details. Only those interfaces actually implemented in connected hardware can be used.

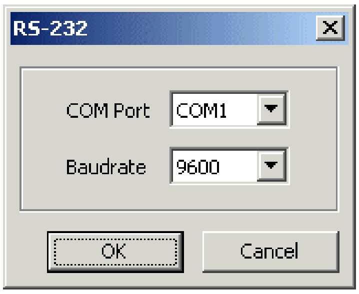
NOTE

The USB drivers make the USB interface appear to the software as an additional RS-232 COM port. That port is present only when the Mercury™ USB device is connected and powered up. The baud rate setting must agree with that set on all devices in the network.

CAUTION

Never connect the RS-232-IN and USB connectors of the same controller to a PC at the same time, as damage may result.

9.4.1. 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. If the USB drivers are installed and a Mercury™ Class controller with USB interface is connected and powered up, the USB interface will appear as an additional COM port.
- `Baud Rate`: The baud rate of the interface. Default value is 9600 as shown. The settings here and on the controller hardware should match.

10. Mercury™ Class Commands

10.1. Functions

- BOOL **Mercury_BRA** (int ID, const char* szAxes, BOOL *pbValarray)
- BOOL **Mercury_CST** (int ID, const char* *szAxes*, const char * *names*)
- BOOL **Mercury_DEL** (int ID, double dSeconds)
- BOOL **Mercury_DFF (**int ID, const char* szAxes, const double * pdValarray)
- BOOL **Mercury_DFH** (int ID, const char* szAxes)
- BOOL **Mercury_DIO** (int ID, const char* szChannels, BOOL *pbValarray)
- BOOL **Mercury_GcsCommandset** (int ID, char* const szCommand)
- BOOL **Mercury_GcsGetAnswer** (int ID, char* szAnswer, const int bufsize)
- BOOL **Mercury_GcsGetAnswerSize** (int ID, int* iAnswerSize)
- BOOL **Mercury_GetInputChannelNames**(int ID, char* szBuffer, int maxlen);
- BOOL **Mercury_GetOutputChannelNames**(int ID, char* szBuffer, int maxlen);
- BOOL **Mercury_GetRefResult**(int ID, const char* szAxes, int* pnResult)
- BOOL **Mercury_GOH** (int ID, const char* szAxes)
- BOOL **Mercury_HLT** (int ID, const char* szAxes)
- BOOL **Mercury_INI** (int ID, const char* szAxes)
- BOOL **Mercury_IsMoving** (const int ID, const char* szAxes, BOOL *pbValarray)
- BOOL **Mercury_IsRecordingMacro** (int ID, BOOL *pbRecordingMacro)
- BOOL **Mercury_IsReferenceOK** (int ID, const char* szAxes, BOOL *pbValarray)
- BOOL **Mercury_IsReferencing** (int ID, const char* szAxes, BOOL *pbIsReferencing)

- BOOL **Mercury_IsRunningMacro** (int ID, BOOL *pbRunningMacro)
- **BOOL Mercury_JDT** (int ID, const int* iJoystickIDs, const int* iValarray, int iArraySize)
- BOOL **Mercury_JON** (int ID, const int* iJoystickIDs, const BOOL* pbValarray, int iArraySize)
- BOOL **Mercury_MAC_BEG** (int ID, const char *szName)
- BOOL **Mercury_MAC_DEL** (int ID, const char *szName)
- BOOL **Mercury_MAC_END** (int ID)
- BOOL **Mercury_MAC_NSTART** (int ID, const char *szName, int nrRuns)
- BOOL **Mercury_MAC_START** (int ID, const char *szName)
- BOOL **Mercury_MEX** (int ID, const char *szCondition)
- BOOL **Mercury_MNL** (int ID, const char* szAxes)
- BOOL **Mercury_MOV** (int ID, const char* szAxes, double *pdValarray)
- BOOL **Mercury_MPL** (int ID, const char* szAxes)
- BOOL **Mercury_MVR** (int ID, const char* szAxes, double *pdValarray)
- int* pnDelay)
- BOOL **Mercury_POS** (int ID, const char* szAxes, double *pdValarray)
- BOOL **Mercury_qBRA** (int ID, char *axes, int maxlen)
- BOOL **Mercury_qCST** (int ID, const char* szAxes, char *names, int maxlen)
- BOOL **Mercury_qDFF (**int ID, const char* szAxes, double * pdValarray)
- BOOL **Mercury_qDFH** (int ID, const char* szAxes, double *pdValarray)
- BOOL **Mercury_qDIO** (int ID, const char* szChannels, BOOL *pbValarray)
- BOOL **Mercury_qERR** (int ID, int *pError)
- BOOL **Mercury_qHLP** (int ID, char *buffer, int maxlen)
- BOOL **Mercury_qIDN** (int ID, char *buffer, int maxlen)
- BOOL **Mercury_qJAX** (int ID, const int* iJoystickIDs, const int* iAxesIDs, int iArraySize, char* szAxesBuffer, int iBufferSize)
- BOOL **Mercury_qJON** (int ID, const int* iJoystickIDs, BOOL* pbValarray, int iArraySize)
- BOOL **Mercury_qLIM** (int ID, const char* szAxes, BOOL *pbValarray)
- BOOL **Mercury_qMAC** (int ID, char *szName, char *szBuffer, int maxlen)
- BOOL **Mercury_qMOV** (int ID, const char* szAxes, double *pdValarray)
- BOOL **Mercury_qNLM** (int ID, const char* szAxes, double *pdValarray)
- BOOL **Mercury_qONT** (int ID, const char* szAxes, BOOL *pbValarray)
- BOOL **Mercury_qPLM** (int ID, const char* szAxes, double *pdValarray)
- BOOL **Mercury_qPOS** (int ID, const char* szAxes, double *pdValarray)
- BOOL **Mercury_qREF** (int ID, const char* szAxes, BOOL *pbValarray)
- BOOL **Mercury_qRON** (int ID, const char* szAxes, BOOL *pbValarray)
- BOOL **Mercury_qSAI** (int ID, char *axes, int maxlen)
- BOOL **Mercury_qSAI_ALL** (int ID, char * axes, int maxlen)
- BOOL **Mercury_qSPA** (int ID, const char* szAxes, const int *iCmdarray, double *dValarray)
- BOOL **Mercury_qSRG**(int ID, const char* szAxes, const int* iCmdarray, int* iValarray)
- BOOL **Mercury_qSVO** (int ID, const char* szAxes, BOOL *pbValarray)
- BOOL **Mercury_qTAC** (int *ID*, int * *pnNr*)
- BOOL **Mercury_qTAV**(int ID, int nChannel, double* pdValue)
- BOOL **Mercury_qTIO** (int ID, int* pNr)
- BOOL **Mercury_qTMN** (int ID, const char* szAxes, double *pdValarray)
- BOOL **Mercury_qTMX** (int ID, const char* szAxes, double *pdValarray)
- BOOL **Mercury_qTNJ** (int ID, int* pnNr);
- BOOL **Mercury_qTVI** (int ID, char *axes, const int maxlen)
- BOOL **Mercury_qVEL** (int ID, const char* szAxes, double *valarray)
- BOOL **Mercury_qVER** (int ID, char *buffer, const int maxlen)
- BOOL **Mercury_qVST** (int *ID*, char * *buffer*, int *maxlen*)
- BOOL **Mercury_REF** (int ID, const char* szAxes)
- BOOL **Mercury_RON** (int ID, const char* szAxes, BOOL *pbValarray)
- BOOL **Mercury_SAI** (int ID, const char* szOldAxes, const char* szNewAxes)
- BOOL **Mercury_SAV** (int ID, const char* szAxes)
- BOOL **Mercury_SPA** (int ID, const char* szAxes, int *iCmdarray, double *dValarray)
- BOOL **Mercury_STP** (int ID)
- BOOL **Mercury_SVO** (int ID, const char* szAxes, BOOL *pbValarray)
- BOOL **Mercury_VEL** (int ID, const char* szAxes, double *valarray)
- BOOL **Mercury_WAC** (int ID, const char *szCondition)

10.2. Detailed Description

These functions encapsulate the GCS ASCII commands supported by Mercury™ Class controllers and provide some shortcuts to make the work with these controllers easier. See “Function Calls“ (p. 11) for some general notes about the parameter syntax. “Types Used in PI Software“ (p. 11) will give you some general information about the syntax of most commands.

NOTE

Keep in mind that a Network of Mercury™ Class controllers chained together and connected to a single host PC interface is handled as single a multi-axis controller by the DLL. Each axis has its own Mercury™ Class controller and the DLL addresses commands for that axis to that controller.

10.3. Function Documentation

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

Corresponding GCS command: BRA

Set brake state for *szAxes* to on (**TRUE)** or off (**FALSE)**. Factory power-up default state for the brake control line is in the “Brake ON” state. INI command sets brake OFF.

Arguments:

iId ID of controller network
szAxes string with axes
pbValarray modes for the specified axes, **TRUE** for on, **FALSE** for off

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: `CLR`

Clear status of *szAxes*.

Arguments:

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

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_CST** (int ID, const char* *szAxes*, const char * *names*)

Corresponding command: `CST`

Set the types of the stages connected to *szAxes*. The individual names must be separated by a line-feed character in the string, rendered by "\n" in the following C source code example: "M-505.1PD\nM-505.2PD".

Arguments:

ID ID of controller network
szAxes identifiers of the stages, if "" or **NULL** all axes are affected
names string with stage-type names separated by line-feed characters ("\n" in C literals)

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: `DEL`

Delay the controller for *dmSeconds* milliseconds.

Note:

The delay will only affect the controller network, the function will return immediately! Commands sent to the controller network during the delay will be queued.

Arguments:

ID ID of controller network
dmSeconds time in milliseconds

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_DFF** (int *ID*, const char* *szAxes*, const double * *pdValarray*)

Corresponding GCS command: `DFF`

Defines a scale factor by which to divide the basic physical units to get the units to use for *szAxes*, e.g. a factor of 25.4 converts the basic physical units of millimeters of all axes in *szAxes* to inches. See also Section 11.3 on p. 45.

Arguments:

iId ID of controller network
szAxes string with axes
pdValarray factors for the axes

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: `DFH`

Makes current positions of *szAxes* the new home position

Arguments:

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

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_DIO** (int ID, const char* *szChannels*, BOOL * *pbValarray*)

Corresponding command: `DIO`

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

Parameters:

ID ID of controller network
szChannels string with digital output channel identifiers; Mercury_GetOutputChannelNames can be used to retrieve the channel names valid for Mercury_DIO

pbValarray array containing the states of specified digital output channels, **TRUE** for "HIGH", **FALSE** for "LOW"

Returns:

TRUE if successful, **FALSE** otherwise

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

Sends a GCS command to the controller network.

Arguments:

ID ID of controller network
szCommand the GCS command as string.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_GcsGetAnswer** (int ID, char* *szAnswer*, const int *bufsize*)

Gets the answer to GCS command (see **Mercury_GcsCommandset**() p. 24).

Arguments:

ID ID of controller network
szAnswer the buffer to receive the answer.
Bufsize the size of the buffer for the answer.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_GcsGetAnswerSize** (int ID, int* *pnAnswerSize*)

Gets the size of the answer to a GCS command (**Mercury_GcsCommandset**() (p. 24)).

Arguments:

ID ID of controller network
pnAnswerSize pointer to integer to receive the size of the next answer.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_GetInputChannelNames** (int *ID*, char **szBuffer*, int *maxlen*)

Get valid single-character identifiers for installed digital input channels. Each character in the returned string is the valid channel identifier of an installed digital input channel. For a Mercury™ Class controller network, the string contains 4 characters for each connected axis (see Section 1.3.2 for details)..

Call Mercury_qDIO() to get the states of the digital inputs.

Parameters:

ID ID of controller network
szBuffer buffer to receive the identifier string
maxlen size of *szBuffer*, must be given to avoid buffer overflow

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_GetOutputChannelNames** (int *ID*, char **szBuffer*, int *maxlen*)

Get valid single-character identifiers for installed digital output channels. Each character in the returned string is the valid channel identifier of an installed digital output channel. For a Mercury™ Class controller network, the string contains 4 characters for each connected axis (see Section 1.3.2 for details). Call Mercury_DIO() using these IDs to set the states of the outputs.

Parameters:

ID ID of controller network
szBuffer buffer to receive the identifier string
maxlen size of *szBuffer*, must be given to avoid buffer overflow

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_GetRefResult** (int *ID*, const char* *szAxes*, int * *pnResult*)

Get results of last call to **Mercury_REF**()(p. 39), **Mercury_MNL**() (p. 30) or **Mercury_MPL**() (p. 30). If still referencing or no reference move was started since startup of library, the result is 0. Call **Mercury_qREF**() (p. 35) to see which axes have a reference switch. **Mercury_REF**() can be used only for axes with reference switches, **Mercury_MNL**() (p. 30) and **Mercury_MPL**() (p. 30) for axes with limit switches. Call **Mercury_IsReferencing**() to find out if there are axes (still) referencing.

Parameters:

ID ID of controller network
szAxes string with axes, if "" or **NULL**, result refers to all axes.
pnResult pointer to array of integers to receive result: 1 if successful, 0 if reference move failed, has not finished yet, or axis does not have the required switch

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: `GOH`

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

Arguments:

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

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: `HLT`

Halt motion of *szAxes* smoothly. Does not work for Mercury_MNL, Mercury_MPL or Mercury_REF motion (use **Mercury_ EmergencyStop**(), p. **Fehler! Textmarke nicht definiert.** instead); after axis stops, target is set to current position. Sets error code 10, whether any motion is stopped or not.

Arguments:

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

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: `INI`

Initialize *szAxes*: resets motion control chip for the axis, sets referenced state to "not referenced", sets the brake control line in the “Brake OFF” state, and if axis was under joystick control, disables the joystick.

Arguments:

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

Returns:

TRUE if successful, **FALSE** otherwise

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

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

Arguments:

ID ID of controller network
szAxes string with axes, if "" or **NULL** all axes are affected.
pbValarray pointer to array to receive statuses of the axes

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_IsRecordingMacro** (int ID, BOOL * *pbRecordingMacro*)

Check if controller is currently recording a macro.

Note:

With Mercury™ Class controllers with native software, Macro recording mode is a state of the library only. See “Macro Storage on Controller,” beginning on p. 46 for more details

Arguments:

ID ID of controller network
pbRecordingMacro pointer to boolean to receive answer: **TRUE** if recording a macro, **FALSE** otherwise

Returns:

TRUE if successful, **FALSE** otherwise

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

Check the reference state of the given axes. Call **Mercury_qREF**() (p. 35) to find out which axes have a reference switch. Axes with a reference switch can be referenced with **Mercury_REF**() (p. 39); axes with limit switches with **Mercury_MNL**() (p. 30) or **Mercury_MPL**() (p. 30).

Arguments:

ID ID of controller network
szAxes string with axes, if "" or **NULL** all axes are queried.
pbValarray pointer to boolean array to receive answers: **TRUE** if the axis is referenced-, **FALSE** if not

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_IsReferencing** (int ID, const char* *szAxes*, BOOL * *pbIsReferencing*)

Check if axis is busy referencing.

Note:

If you do not specify any axis, you will get back only one `BOOL`. It will be **TRUE** if the controller is referencing any axis.

Arguments:

ID ID of controller network
szAxes string with axes, if "" or **NULL** single value is returned: TRUE if any axis is being referenced.
pbIsReferencing pointer to boolean array to receive statuses of axes or of the controller, **TRUE** if referencing, **FALSE** otherwise

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: `#8`

Check if controller is currently running a macro

Arguments:

ID ID of controller network
pbRunningMacro pointer to boolean to receive answer: **TRUE** if a macro is running on at least one of the devices in the network, **FALSE** otherwise

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_JDT** (int ID, const int* iJoystickIDs, const int* piValarray, int iArraySize)

Corresponding command: `JDT`

Load pre-defined joystick response table. The table type can be either 1 for linear or 3 for cubic response curve.
The cubic curve offers more sensitive control around the middle position and less sensitivity close to the maximum velocity.

Arguments:

ID ID of controller network
iJoystickIDs array with device numbers of motion-axis controllers, each with a joystick axis attached
piValarray pointer to array with table types for the corresponding joystick axes
iArraySize size of arrays

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: `JON`

Enable/disable direct joystick control for given motion-controller axes. To enable, set the corresponding entry in pbValarray to TRUE. The motion-controller axes are identified by the device number of the Mercury™ Class controller to which the joystick axis is connected (see p. 8). See the controller User Manual for Device Number setting; typically 4 DIP switches are used to set a negative-logic, binary value one less than the device number.

Do not enable axes with no physical joystick connected, as uncontrolled motion could occur. The C-862 Mercury™ DC Motor Controller does not have a joystick port.

Arguments:

ID ID of controller network
iJoystickIDs array with device numbers of devices having a directly connected joystick axis
pbValarray pointer to array with joystick enable states for the specified motion-axis controllers (0 for deactivate, 1 for activate)
iArraySize size of arrays

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_MAC_BEG** (int ID, char * *szName*)

Corresponding command: `MAC BEG`

Put the DLL in macro recording mode. See "Macro Storage on Controller," beginning on p. 46 for details. This function sets a flag in the library and effects the operation of other functions. Function will fail if already in recording mode. If successful, the commands that follow become part of the macro, so do not check error state unless FALSE is returned.

Arguments:

ID ID of controller network
szName name under which macro will be stored in the controller, must of the form *a*MC0*nn* where *a* is the axis designation of the axis controlled by the controller on which the macro is to be stored and *nn* is the ID number for the macro, 0 to 31 (Macro 0 is executed on power up or reset, whether there is a PC connected or not).

Returns:

TRUE if successful, **FALSE** otherwise

Errors:

PI_IN_MACRO_MODE if a macro is already being recorded

BOOL **Mercury_MAC_DEL** (int ID, char * *szName*)

Corresponding command: `MAC DEL`

Delete macro with name *szName*. To find out what macros are available call **Mercury_qMAC**() (p. 34). See "Macro Storage on Controller," beginning on p. 46 for more details

Arguments:

ID ID of controller network
szName name of the macro to delete

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_MAC_END** (int ID)

Corresponding command: `MAC END`

Take the DLL out of macro recording mode. This function resets a flag in the library and effects the operation of certain other functions. Function will fail if the DLL is not in recording mode. See "Macro Storage on Controller," beginning on p. 46 for more details

Arguments:

ID ID of controller network

Returns:

TRUE if successful, **FALSE** otherwise

Errors:

PI_NOT_IN_MACRO_MODE the controller was not recording a macro

BOOL **Mercury_MAC_NSTART** (int ID, char * *szName*, int *nrRuns*)

Corresponding command: `MAC START`

Start macro with name *szName*. The macro is repeated *nrRuns* times. To find out what macros are available call **Mercury_qMAC**() (p. 34). See "Macro Storage on Controller," beginning on p. 46 for more details.

Arguments:

ID ID of controller network
szName string with name of the macro to start
nrRuns nr of runs

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_MAC_START** (int ID, char * *szName*)

Corresponding command: `MAC START`

Start macro with name *szName*. To find out what macros are available call **Mercury_qMAC**() (p. 34). See "Macro Storage on Controller," beginning on p. 46 for more details.

Arguments:

ID ID of controller network
szName string with name of the macro to start

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_MEX** (int ID, char * *szCondition*)

Corresponding command: `MEX`

Stop Macro EXecution due to a given condition of the following type: one given value is compared with a queried value according to a given rule.

Can only be used in macros.

When the macro interpreter accesses this command the condition is checked. If it is true the current macro is stopped, otherwise macro execution continues with the next line. If the condition is fulfilled later, it has no effect.

Valid conditions are

- DIO?, but only the digital I/O channels of the Mercury™ on which the macro is stored can be queried
- JBS?, but only the button 1 associated with the joystick axis connected to the controller on which the macro is stored can be queried

(See "Macro Storage on Controller," p. 46)

Examples:

Mercury_MEX(ID, "DIO? A = 1");

Mercury_MEX(ID, "JBS? 4 1 = 1");

Arguments:

ID ID of controller network
szCondition string with condition to evaluate

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: `MNL`

For each of the axes in *szAxes* in turn, reset soft limits and home position, move the axis to its negative limit switch and back until the limit switch disengages, set the position counter to the minimum position value and set the reference state to "referenced". This can be used to reference axes without reference switches. **Mercury_MNL()** returns before the controller has finished. Call **Mercury_IsReferencing**() (p. 27) to find out if the axes are still moving and **Mercury_GetRefResult**() (p. 25) to get the results of the referencing move. The controller will be "busy" while referencing, so most other commands will cause a **PI_CONTROLLER_BUSY** error. Use **Mercury_STP**() (p. **Fehler! Textmarke nicht definiert.**) to stop referencing motion.

Arguments:

ID ID of controller network
szAxes axes to move.

Returns:

TRUE if successful, **FALSE** otherwise

Errors:

PI_UNKNOWN_AXIS_IDENTIFIER *szAxes* contains an invalid axis identifier

BOOL **Mercury_MOV** (int ID, const char* *szAxes*, double * *pdValarray*)

Corresponding command: `MOV`

Move *szAxes* to absolute position.

Arguments:

ID ID of controller network
szAxes string with axes
pdValarray pointer to array with target positions for the axes

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: `MPL`

For each of the axes in *szAxes* in turn, reset soft limits and home position, move the axis past its positive limit switch and back until the limit switch disengages, set the position counter to the maximum position value, and set the reference state to "referenced" . This can be used to reference axes without reference switches. **Mercury_MPL()** returns before the controller has finished. Call **Mercury_IsReferencing**() (p. 27) to find out if the axes are still moving and **Mercury_GetRefResult**() (p. 25) to get the results of the referencing move. The controller will be "busy" while referencing, so most other commands will cause a **PI_CONTROLLER_BUSY** error. Use **Mercury_STP**() (p. 24) to stop referencing motion.

Arguments:

ID ID of controller network
szAxes axes to move.

Returns:

TRUE if successful, **FALSE** otherwise

Errors:

PI_UNKNOWN_AXIS_IDENTIFIER `cAxis` is no valid axis identifier

BOOL **Mercury_MVR** (int ID, const char* *szAxes*, double * *pdValarray*)

Corresponding command: `MVR`

Move *szAxes* relatively.

Arguments:

ID ID of controller network
szAxes string with axes
pdValarray pointer to array with distances to move in physical units

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_POS** (int ID, const char* *szAxes*, double * *pdValarray*)

Corresponding command: POS

Sets absolute positions (position counters) for given axes. Reference mode for the axes must be OFF. No motion occurs. See Mercury_RON() for a detailed description of reference mode and how to turn it on and off. For stages with neither reference nor limit switch, reference mode is automatically OFF.

Note that when the actual position is incorrectly set with this command, stages can be driven into the limit switch when moving to a position which is thought to be within the travel range of the stage, but actually is not.

Arguments:

ID ID of controller network
szAxes string with axes
pdValarray pointer to array with absolute positions for the specified axes, in physical units

Returns:

TRUE if successful, **FALSE** otherwise

Errors:

PI_CNTR_CMD_NOT_ALLOWED_FOR_STAGE if the reference mode for any of the given axes is ON

BOOL **Mercury_qBRA** (int ID, char * *szBuffer*, int*maxlen*)

Corresponding GCS command: BRA?

Get axes with brakes.

Arguments:

iId ID of controller network
szBuffer buffer to store the read in string
maxlen size of *buffer*, must be given to avoid a buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qCST** (int ID, const char* *szAxes*, char * *names*, const int *maxlen*)

Corresponding command: CST?

Get the type names of the connected stages *szAxes*. The individual names are preceded by the axis identifier and an equals sign ("=") and followed by an ASCII line-feed character For example A=M-714.00.1PDLF

B=M-511.HDLF.

Arguments:

ID ID of controller network
szAxes identifiers of the stages, if "" or **NULL** all axes are queried
names buffer to receive the list of names read in from controller, lines are separated by line-feeds
maxlen size of *name*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qDFF** (int *ID*, const char* *szAxes*, double * *pdValarray*)

Corresponding GCS command: `DFF?`

Get scale factors for *szAxes* set with **Mercury_DFF**()

Arguments:

iId ID of controller network
szAxes string with axes, if "" or **NULL** all axes are queried.
pdValarray pointer to array to receive factors of the axes

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qDFH** (int ID, const char* *szAxes*, double * *pdValarray*)

Corresponding command: `DFH?`

Get displacement of the home position from its default for *szAxes* in physical units.

Arguments:

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

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qDIO** (int ID, const char* *szChannels*, BOOL * *pbValarray*)

Corresponding command: `DIO?`

Get the states of *szChannels* digital input channel(s).

Parameters:

ID ID of controller network
szChannels string with digital input channel identifiers, if "" or **NULL** all channels are queried.
pbValarray pointer to array to receive the states of digital input channels: **TRUE** if "HIGH", **FALSE** if "LOW"

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qERR** (int ID, int * *pError*)

Corresponding command: `ERR?`

Get the error state of the controller. It is safer to call **Mercury_GetError**()(p. 18) because this will check the internal error state of the library first.

Arguments:

ID ID of controller network
pnError pointer to integer to receive error code of the controller

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qHLP** (int ID, char * *buffer*, const int *maxlen*)

Corresponding command: `HLP?`

Read in the help string of the controller. The answer is quite long (up to 3000 characters) so be sure to provide enough space!.

Arguments:

ID ID of controller network
buffer buffer to receive the string read in from controller, lines are separated by line-feed characters.
maxlen size of *buffer*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qIDN** (int ID, char * *buffer*, const int *maxlen*)

Corresponding command: `*IDN?`

Get identification string of the controller.

Arguments:

ID ID of controller network
buffer buffer to receive the string read in from controller; contains controller hardware full name, firmware version and date
maxlen size of *buffer*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qJAX** (int ID, const int* iJoystickIDs, const int* iAxesIDs, int iArraySize, char* szAxesBuffer, int iBufferSize)

Corresponding command: `JAX?`

Reports correspondence between joystick port numbers (device numbers) and axis identifiers for axes with joystick ports.

Arguments:

ID ID of controller network
iJoystickIDs array with device numbers of devices having a directly connected joystick axis
iAxesIDs array with axis IDs of the joystick axes (must be 1 for C-663, which only has 1 joystick axis per device)
iArraySize size of arrays
buffer buffer to receive the string read in from controller; will contains axis IDs of axes associated with corresponding joystick axis
maxlen size of *buffer*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: `JON?`

Gets joystick enable/disable states for given motion-controller axes. The joystick axes are identified by the device number of the Mercury™ Class controller to which they are connected.(see p. 8) See the controller User Manual for Device Number setting; typically 4 DIP switches are used to set a negative-logic, binary value one less than the device number . See also Mercury_JON()

Arguments:

ID ID of controller network
iJoystickIDs array with device numbers of devices having a directly connected joystick axis
pbValarray pointer to array to receive the joystick-axis enable states of the specified motion-controller axes (0 for deactivated, 1 for activated)
iArraySize size of arrays

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: `LIM?`

Check if the given axes have limit switches

Arguments:

ID ID of controller network
szAxes string with axes, if "" or **NULL** all axes are queried.
pbValarray pointer to array to receive the limit-switch info: **TRUE** if axis has limit switches, **FALSE** if not

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qMAC** (int ID, char * *szName*, char * *szBuffer*, const int *maxlen*)

Corresponding command: `MAC?`

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

Arguments:

ID ID of controller network
szName string with name of the macro to list
szBuffer buffer to receive the string read in from controller, lines are separated by line-feed characters
maxlen size of *buffer*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qMOV** (int ID, const char* *szAxes*, double * *pdValarray*)

Corresponding command: `MOV?`

Read the commanded target positions for *szAxes*.

Arguments:

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

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: `ONT?`

Check if *szAxes* have reached target position.

Arguments:

ID ID of controller network
szAxes string with axes, if "" or **NULL** all axes are queried and a separate answer provided for each.
pdValarray pointer to array to be filled with current on-target status of the axes

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qPOS** (int ID, const char* *szAxes*, double * *pdValarray*)

Corresponding command: `POS?`

Get the positions of *szAxes*.

Arguments:

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

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: `REF?`

Check if the given axes have reference switches

Arguments:

ID ID of controller network
szAxes string with axes, if "" or **NULL** all axes are queried.
pbValarray pointer to array for answers: **TRUE** if axis has a reference switch, **FALSE** if not

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: `RON?`

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

Arguments:

ID ID of controller network
szAxes string with axes, if "" or **NULL** all axes are queried
pbValarray pointer to array to receive reference modes for the specified axes

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qSAI** (int ID, char * *axes*, const int *maxlen*)

Corresponding command: `SAI?`

Get connected axes. Each character in the returned string is an axis identifier for one connected axis.

Arguments:

ID ID of controller network
axes buffer to receive the string read in
maxlen size of *buffer*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qSAI_ALL** (int ID, char * *axes*, int *maxlen*)

Corresponding GCS command: `SAI? ALL`

Get all possible axes, and not only all connected and configured axes as returned by the Mercury_qSAI function. Each character in the returned string is an axis identifier for one possible axis.

Arguments:

iId ID of controller network
axes buffer to store the read in string
maxlen size of *buffer*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qSPA** (int ID, const char* *szAxes*, int * *iCmdarray*, double * *dValarray*)

Corresponding command: SPA?

Read parameters for *szAxes*. For each desired parameter you must specify an axis in *szAxes* and a parameter ID in the corresponding element of *iCmdarray*. See Section 11 on p. 42 for a list of valid parameter IDs.

Arguments:

ID ID of controller network
szAxes axes for each of which a parameter should be read
iCmdarray IDs of parameters
dValarray array to be filled with the values of the parameters

Returns:

TRUE if successful, **FALSE** otherwise

Errors:

PI_INVALID_SPA_CMD_ID *one* of the IDs in *iCmdarray* is not valid

BOOL **Mercury_qSRG** (int ID, const char* *szAxes*, const int * *iCmdarray*, long * *lValarray*)

Corresponding command: SRG?

Read the values of the specifed registers

ID of the parameters can only be 3, which will read in the signal input lines register (byte 2 of the C-663 and byte 4 for the C-862). See the Mercury GCS Commands manual for detailed description of the parameters

.

Arguments:

ID ID of controller network
szAxes axes for each of which a parameter should be read
iCmdarray IDs of parameters
lValarray array to be filled with the values of the registers

Returns:

TRUE if successful, **FALSE** otherwise

Errors:

PI_INVALID_SPA_CMD_ID *one* of the IDs in *iCmdarray* is not valid

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

Corresponding command: SV0?

Get the servo mode for *szAxes*

Arguments:

ID ID of controller network
szAxes string with axes, if "" or **NULL** all axes are queried.
pbValarray pointer to array to receive the servo-modes of the specified axes: **TRUE** for "on", **FALSE** for "off"

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: `TAC?`

Get the number of installed analog channels.

Parameters:

ID ID of controller network
pnNr pointer to `int` to receive the number of installed boards

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qTAV** (int *ID*, int *nChannel*, double * *pdValue*)

Corresponding command: `TAV?`

Read analog input.

Parameters:

ID ID of controller network
nChannel index of channel to use (see Section 1.3.2)
pdValue pointer to `double` for storing the value read from analog input

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qTIO** (int ID, int * *pnINr*, int * *pnONr*)

Corresponding command: `TIO?`

Get the number of digital input and output channels installed.

Arguments:

ID ID of controller network
pnINr pointer to `int` to receive the number of digital input channels installed
pnONr pointer to `int` to receive the number of digital output channels installed

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qTMN** (int ID, const char* *szAxes*, double * *pdValarray*)

Corresponding command: `TMN?`

Get the low end of travel range of *szAxes* in physical units and relative to the current home position.

Arguments:

ID ID of controller network
szAxes string with axes, if "" or **NULL** all axes are queried.
pdValarray pointer to array to be filled with minimum positions of the axes

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qTMX** (int ID, const char* *szAxes*, double * *pdValarray*)

Corresponding command: `TMX?`

Get the high end of the travel range of *szAxes* in physical units and relative to the current home position.

Arguments:

ID ID of controller network
szAxes string with axes, if "" or **NULL** all axes are queried.
pdValarray pointer to array to be filled with maximum positions of the axes

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: `TNJ?`

Get the number of joysticks. Note: the software can not determine if a joystick is actually connected to a C-663. This is the maximum possible number of joysticks that can be connected to the network..

Parameters:

ID ID of controller network
pnNr pointer to `int` to receive the number of joysticks

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qTVI** (int ID, char * *axes*, const int *maxlen*)

Corresponding command: `TVI?`

Get list of all characters that can be used as axis identifiers. Each character in the returned string could be used as a valid axis identifier after being assigned with Mercury_SAI().

Arguments:

ID ID of controller network
axes buffer to receive the string read in
maxlen size of *buffer*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qVEL** (int ID, const char* *szAxes*, double * *valarray*)

Corresponding command: `VEL?`

Get the velocity settings of *szAxes*. This is the velocity set to be used for moves.

Arguments:

ID ID of controller network
szAxes string with axes, if "" or **NULL** all axes are queried.
pdValarray pointer to array to be filled with the velocities of the axes

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qVER** (int ID, char * *buffer*, const int *maxlen*)

Corresponding command: VER?

Get version of the controller firmware.

Arguments:

ID ID of controller network
buffer buffer to receive the string read in
maxlen size of *buffer*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qVST** (int *ID*, char * *buffer*, int *maxlen*)

Corresponding command: VST?

Get the names of stages selectable with Mercury_CST().

Parameters:

ID ID of controller network
buffer buffer to receive the string read in from controller, lines are separated by line-feed characters
maxlen size of *buffer*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: REF

For each of the axes in *szAxes*.turn, reset soft limits and home position, move the axis to its reference switch (passing it if necessary, to approach from the negative side), set the position counter to the minimum position value and set the reference state to "referenced." Each axis must be equipped with a reference switch (use Mercury_qREF() to find out). **Mercury_REF()** returns before the controller has finished. Call **Mercury_IsReferencing**() (p. 27) to find out if the axes are still moving and **Mercury_GetRefResult**() (p. 25) to get the results of the referencing move. The controller will be "busy" while referencing, so most other commands will cause a **PI_CONTROLLER_BUSY** error. Use **Mercury_STP**() (p. ***Fehler! Textmarke nicht definiert.***) to stop reference motion.

Arguments:

ID ID of controller network
szAxes string with axes

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: RON

Sets reference mode for given axes.

If the reference mode of an axis is ON, the axis must be driven to the reference switch (Mercury_REF()) or to a limit switch (using Mercury_MPL() Mercury_MNL()) before any other motion can be commanded.

If reference mode is OFF, no referencing is required for the axis. Only relative moves can be commanded (Mercury_MVR()), unless the controller is informed of the actual position with Mercury_POS(). Afterwards, relative and absolute moves can be commanded.

For stages with neither reference nor limit switch, reference mode is automatically OFF.

Note that when the reference mode is off and the actual position is incorrectly set with Mercury_POS(), stages can be driven into the limit switch when moving to a position which is thought to be within the travel range of the stage, but actually is not.

Arguments:

ID ID of controller network
szAxes string with axes
pbValarray pointer to array to receive the reference modes for the specified axes

Returns:

TRUE if successful, **FALSE** otherwise

Errors:

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

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

Corresponding command: `SAI`

Rename connected axes. Axis designated by the first character in *szOldAxes* will be renamed to first character in `szNewAxes`, etc. with the remaining characters of the two equal-length strings. User can change the "names" of axes with this function. The characters in *szNewAxes* character must not be in use for another existing axis 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 **Mercury_qTVI**() (p. 38). Only the **last** occurrence of an axis identifier in *szNewAxes* will be used to change the name.

Arguments:

ID ID of controller network
szOldAxes old identifiers of the axes
szNewAxes new identifiers of the axes

Returns:

TRUE if successful, **FALSE** otherwise

Errors:

PI_INVALID_AXIS_IDENTIFIER if the characters are not valid
PI_UNKNOWN_AXIS_IDENTIFIER if *szOldAxes* contains unknown axes
PI_AXIS_ALREADY_EXISTS if one of *szNewAxes* is already in use
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 Mercury_SPA (int ID, const char* szAxes, int * iCmdarray, double * dValarray)

Corresponding command: SPA

Set parameters for *szAxes*. For each parameter you must specify an axis in *szAxes* and a parameter ID in the corresponding element of *iCmdarray*.

Mercury_SPA has two arrays as arguments. The first array has the parameters which have to be modified, the second one the values. If you want to set the velocity (ID=10) to 0.05, the acceleration (ID=11) to 8 and the deceleration (ID=12) to 8, you can use the following code (in C(++) syntax):

```
char szAxes[] = “AAA”;
int cmd[] = {10, 11, 12};
double values[] = {0.05, 8, 8};
Mercury_SPA(id, szAxes, cmd, values);
```

szAxes = "AAA"	cmd = {10, 11, 12}	values = {0.05, 8, 8}
szAxes[0] = 'A'	cmd[0] = 10	values[0] = 0.05
szAxes[1] = 'A'	cmd[1] = 11	values[1] = 8
szAxes[2] = 'A'	cmd[2] = 12	values[2] = 8

Note:

If the same axis has the same parameter ID more than once, only the **last** value will be set. For example `Mercury_SPA(id, "AAA", {10, 10, 12}, {0.06, 0.05, 9})` will set the velocity of 'A' to 0.05 and the deceleration to 9.

Arguments:

ID ID of controller network
szAxes axis for which the parameter should be set
iCmdarray IDs of parameters
dValarray array with the values for the parameters

Returns:

TRUE if successful, **FALSE** otherwise

Errors:

PI_INVALID_SPA_CMD_ID *one* of the IDs in *iCmdarray* is not valid

BOOL **Mercury_STP** (int ID)

Corresponding command: STP

Stop all axes. This includes motion of all axes (Mercury_MOV, Mercury_MVR), referencing motion (Mercury_MNL, Mercury_MPL, Mercury_REF) and macros.

Sets error code to 10, whether any axis was in motion or not.

Arguments:

ID ID of controller network

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: SVO

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

Arguments:

ID ID of controller network

szAxes string with axes
pbValarray pointer to boolean array with servo-modes for the specified axes, **TRUE** for "on", **FALSE** for "off"

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_VEL** (int ID, const char* *szAxes*, double * *valarray*)

Corresponding command: VEL

Set the velocities to use for moves of *szAxes*.

Arguments:

ID ID of controller network
szAxes string with axes
pdValarray pointer to array with velocity settings for the axes

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_WAC** (int ID, char * *szCondition*)

Corresponding command: WAC

WAit until a given **C**ondition of the following type occurs: one given value is compared with a queried value according to a given rule.

Can only be used in macros.

When the macro interpreter accesses this command the condition is checked. If it is true the current macro is stopped, otherwise macro execution continues with the next line. If the condition is fulfilled later it has no effect.

Valid conditions are ONT? and DIO?, but only the digital I/O channels or the axis of the Mercury™ on which the macro is stored can be queried (see Section 12)

Exampe: Mercury_WAC(ID, “ONT? A = 1”);

Arguments:

ID ID of controller network
szCondition string with condition to evaluate

Returns:

TRUE if successful, **FALSE** otherwise

11. Motion Parameters Overview

11.1. Parameter Handling

CAUTION !

The parameters listed in Section 11.2 are hardware-specific. Incorrect values may lead to improper operation or damage of your hardware! Change settings only after consultation with PI.

Most of the parameters listed below describe the physical properties and limits of a stage. They can be changed by several functions, but modifying them imprudently could cause damage to the stage. So please handle these parameters with care.

Generally, parameters should only be changed in real special cases and only after consultation with PI, especially the servo-loop parameters.

With Mercury_SPA? (p. 36) you can obtain a list of the current parameter values in RAM.

11.2. Parameter List

Parameter numbers in italics apply to C-663, those in bold to C-862

1	P-Term	0 to 32767	-	
2	I-Term	0 to 32767	-	
3	D-Term	0 to 32767	-	
4	I-Limit	0 to 32767	-	
8	Maximum position error	0 to 32767	Counts	
10	Maximum allowed velocity	> 0	Physical units	This parameter is a maximum limit and not the current velocity. By default the current velocity is half the maximum allowed velocity. To change the current velocity use the VEL() command.
11	Maximum allowed acceleration		Physical units	
14	Numerator of the counts per physical unit factor	1 to 2147483647	-	factor = num./denom. This factor includes the physical transmission ratio and the resolution of the stage. Note: To customize your physical unit use parameter 18 instead.
15	Denominator of the counts per physical unit factor	1 to 2147483647	-	factor = num./denom. This factor includes the physical transmission ratio and the resolution of the stage. Note: To customize your physical unit use parameter 18 instead.
17	Invert the direction	-1 to invert the direction, else 1	-	
18	Scaling factor	-1.79769313486231E308 to 1.79769313486231E308	-	This factor can be used to change the physical unit of the stage, e.g. a factor of 25.4 converts a physical unit of mm to inches. It is recommended to use the **DFF**() command to change this factor.
19	Rotary stage	1 = rotary stage, else 0	-	
20	Stage has a reference	1 = the stage has a reference, else 0	-	
21	Maximum travel range in positive direction	0 to 2147483647	Physical units	
22	Value at reference position	-2147483647 to 2147483647	Physical units	

23	Distance from the negative limit to the reference position	-2147483647 to 2147483647	Physiccal units	
24	Axis limit mode	0 = positive limit switch active high (pos-HI), negative limit switch active high (neg-HI) 1 = positive limit switch active low (pos-LO), neg-HI 2 = pos-HI, neg-LO 3 = pos-LO, neg-LO	-	
25	Stage type	0 = DC motor 2 = Voice coil 4 = Piezo motor	-	Wrong stage type will cause malfunction.
48	Maximum travel range in negative direction	-2147483647 to 2147483647	Physiccal unit	
49	Invert the reference	1 = invert the reference, else 0	-	
60	Stage name	maximum 15 characters	-	
64	Hold Current (HC native command) in mA			
65	Drive Current (DC native command) in mA			
66	Hold Time (HT native command) in ms			
67	max current, max. value that DC and HC can have, in mA			

11.3. Transmission Ratio and Scaling Factor

The physical unit used for the stages (i.e. for the axes of the controller) results from the following interrelation of some stage parameters:

$$PU = \left(Cnt / \frac{CpuN}{CpuD} \right) \times SF$$

$$Cnt = \left(PU / SF \right) \times \frac{CpuN}{CpuD}$$

Name	Number*	Description
PU	-	Physical Unit
Cnt	-	Counts
CpuN	14	Numerator of the counts per physical unit factor
CpuD	15	Denominator of the counts per physical unit factor
SF	18	Scaling factor**

*Number means the parameter ID in Mercury_SPA (p. 41) and Mercury_qSPA (p. 36) and in the list in Section 11.2.

**See also Mercury_DFF (p. 23).

The "Counts per physical unit factor" which results from parameter 14 divided by parameter 15 includes the physical transmission ratio and the resolution of the stage.

CAUTION

To customize the physical unit of a stage do not change parameter 14 and parameter 15 but use Mercury_DFF (p. 23) instead. Although Mercury_DFF has the same effect as changing parameter 18 with Mercury_SPA, you should only use Mercury_DFF and not Mercury_SPA to modify the scaling factor.

Example: If you set with Mercury_DFF a value of 25.4 for an axis, the physical unit for this axis is converted from mm to inches.

12. Macro Storage on Controller

Up to 32 macros can be stored in non-volatile memory on each Mercury™ Class controller. Macros are stored in the command language of the controller. With present firmware, this is the Mercury™ native command set.

12.1. Features and Restrictions

The native-command macro storage facility has the following features, which result in certain restrictions:

- Each macro can contain up to 16 such commands
- The macros are identified by numbers 0 to 31
- Macro 0, if defined, is the autostart macro, which is executed automatically upon power-up or reset
- Macros are executed on the controller where they are stored, so commands in a macro may address only the axis and/or I/O channels associated with that controller (there is no command-interface communication between controllers). Interaction between separate axes is conceivable only through suitable programming and hardwiring of I/O lines
- The position values stored in the macros are in counts. This means that a macro may not work properly if run when different stage types are connected to the controller. A different stage could have a very different travel ratio and thus move to a position far different from the one intended.

12.2. Native Macro Recording Mechanism

A macro is stored on the controller by placing it in a compound command beginning with the native command MDn, (define macro n). See the Mercury Native Commands manual for details.

12.3. Macro Translation by the GCS DLL

12.3.1. Macro Creation from GCS

The GCS macro creation mechanism involves placing a GCS controller in macro-recording mode, sending it commands, and then exiting macro recording mode. While in macro-recording mode, the controller neither executes nor responds to commands, but simply stores them in the macro.

In normal operation, the GCS DLL translates GCS-based functions to Mercury™ native commands. The GCS macro-recording mechanism is easily translated to native commands with the use of a macro-recording flag in the DLL. While the flag is set, DLL function calls create native commands as usual but they are saved rather than sent to the controller. When recording is completed (Mercury_MAC_END() function), the saved commands are assembled into a compound command beginning with MD, given a cursory check, and, if they are acceptable, the macro definition compound command is sent to the controller.

Here are some of the implications:

- The DLL may decide not to send the macro to the controller at all. Whether or not the macro was sent can be checked with Mercury_qERR after Mercury_MAC_END(): If the macro was not sent, error -1010 will be set. (Admittedly, the error-description text can be misleading)

- Referencing operations with REF are allowed, because with the Mercury™ native command set it is possible to tell how to move toward or away from the reference switch. Because REF is not implemented as single commands in the native command set, it will occupy more than one command slot in the macro (see examples below).
- A total of only 16 (native) commands may be stored in a macro on a Mercury™ Class controller. That means that when using GCS commands which translate to multiple native commands (e.g. REF, INI), little space may be left for other commands.
- The way in which a GCS function is translated into a native command can depend on the stage connected and how it was referenced. A macro made under one set of conditions will not function properly if run under others[*]. As a result:
  - Macros are only valid for the stage type that was connected when the macro was created.
  - Only relative moves can be used without concern in macros
  - Absolute moves require the axis to have been referenced with exactly the same sequence of referencing commands when the macro is run as when it was created. (Note that having the software save positions at shutdown and restore them from saved values involves RON/POS referencing.)[**]
- The macro names used at the GCS level are assigned using the following strict convention: *a*MC0*nn* where *a* is the current axis designator associated with the controller and *nn* is a two-digit number between 00 and 31.In addition, all the MAC commands take an axis designator as an argument. The macros AMC000, BMC000, etc. (for axes A, B,..., respectively) are the autostart macros, which are executed automatically upon startup or reset of the individual axis controller. The name thus already specifies the axis which the macro addresses.
- Only the following GCS DLL functions are allowable when the macro recording flag is set. Use of a disallowed command will cause the next MAC END to set an error.
  - Mercury_BRA()
  - Mercury_DEL()
  - Mercury_DFH()
  - Mercury_DIO()
  - Mercury_GOH()
  - Mercury_HLT()
  - Mercury_INI() (generates a large number of native commands in the macro, see below)
  - Mercury_IsRecordingMacro()
  - Mercury_MAC START() (macro called must reside on the same controller)

---

[*] For example, position values in millimeters or degrees in GCS motion commands are converted to counts. The count values are calculated when the macro is created using the parameters for the stage configured on the corresponding axis (controller).

[**] Because it is not possible to set the current absolute position to a desired value, but only to 0, the count values in the controller's internal position counter after a GCS move to a given position may be very different depending on how the axis was referenced (with REF, MNL, MPL or a RON/POS combination),

- Mercury_MAC_END() (takes DLL out of Macro Recording Mode)
- Mercury_MEX() with “DIO? <channel> = <b>” as condition
- Mercury_MEX() with “JBS? <joystick> 1 = <b>” as condition
- Mercury_MVR()
- Mercury_REF() (generates a large number of native commands in the macro, see below)
- Mercury_SPA()

  Access to the following SPA parameters by macros is permitted: all others will be ignored:

  - 1: P-Term
  - 2: I-Term
  - 3: D-Term
  - 4: I-Limit
  - 8: Max.Position Error
  - 10: Max. Velocity
  - 11: Max Acceleration (muss >200 sein)
  - 24: Limit Switch Mode
  - 50: No Limit Switch
  - 64: Hold Current (HC native command) in mA
  - 65: Drive Current (DC native command) in mA
  - 66: Hold Time (HT native command) in ms
- Mercury_STP()
- Mercury_SVO()
- Mercury_VEL()
- Mercury_WAC() with “DIO? <channel> = <b>” as condition (where b = 1 or 0 for TRUE, FALSE)
- Mercury_WAC() with “ONT? <axis> = 1” as condition

12.3.2. GCS Listing Stored Macros

When the **Mercury_qMAC()** function is used with a macro name to list the contents of a macro, the native commands stored on the unit are translated back to GCS commands (See the GCS Mercury™ Commands Manual, document MS163E for details), with all the implications that entails.

Functions that cause several native commands to be stored in the macro may not be recognized when the macro is listed, making it possible to see the GCS versions of the individual functions (see INI example below).

The native-command versions can, of course, be manipulated by sending the native commands MD*n*, TM*n,* TZ, etc. (Macro Define, Tell Macro *n*, Tell Macro Zero) with **Mercury_Sendnongcsstring()** (see Mercury Native Commands manual for native command descriptions).

Native commands that have no equivalent in GCS (e.g. FE3) are listed in their original form as follows:

“<non GCS: FE3>”

12.3.3. Macro Translation and Listing Examples

INI

When converted to native commands, INI is separated into all of its separate functions; when the stored macro is listed with MAC? they are shown as a long list of separate commands. From the list it is obvious that when INI is used, not many commands are left before the macro is full. With an M-505.4PD, the dialog can look as follows:

```
>>CST DM-505.4PD
>>ERR?
<<0
>>MAC BEG DMC003
>>INI D
>>MAC END
>>ERR?
<<0
>>MAC? DMC003
<<SPA D50 0
<<SPA D24 0
<<BRA D0
<<SPA D1 200
<<SPA D2 150
<<SPA D3 100
<<SPA D8 2000
<<SPA D4 2000
<<SVO D1
<<VEL D25
<<SPA D11 4000000
<<STP
```

REF

Similarly, REF A, is stored as the following sequence (shown this time in the native command set):

"`SV40000,FE2,WS,MR-40000,WS,FE,WS,SV100000`"

This sequence, when read with MAC?, is recognized by the DLL and translated back to REF A, obscuring the fact that it occupies 8 of the 16 possible command slots. It can thus be seen, that INI and REF will not both fit in the same macro!

MVR

The relative move sizes entered with MVR and converted into counts using the parameters of the currently configured stage before being stored. So, if a macro containing MVR A2 is created with an M-111.2DG configured on axis A and later an M-505.4PD is configured for A with CST, the macro will read out as MVR A 58.2542.

13. Error Codes

The error codes listed here are those of the *PI General Command Set.* As such, some are not relevant to C-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

14. Index