Piezo · Nano · Positioning

MS154E Software Manual

Mercury™ Class

PI_Mercury_GCS_DLL

Release: 1.1.0 Date: 2009-08-11

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

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

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

Moving the NanoWorld | www.pi.ws

First printing 2009-08-11
Document Number MS154E, Eco, Bro, Release 1.1.0
PI_Mercury_GCS_DLL_MS154E.doc

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 PI Mercury GCS DLL

The PI_Mercury_GCS_library 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.

The PI General Command Set (GCS) is the PI standard command set and ensures the compatibility between different PI controllers.

The library is available for the following operating systems:

- **Windows** 2000, XP and Vista: PI Mercury GCS DLL
  See Sections 3, 4 and 5 for more information about PI DLLs.
- **Linux** operating systems (kernel 2.6, GTK 2.0, glibc 2.4): libpi_mercury_gcs.so*.x.x.x* and libpi_mercury_gcs-*x.x.x*.a where *x.x.x* gives the version of the library

NOTES

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

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 PI Mercury GCS DLL user as a single, multi-axis controller and is usually referred to in this manual as a “controller network”.

1.1. 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 GCS 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.

1.2. Units and GCS

1.2.1. 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 0xE and 0xF). 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.

1.2.2. 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.

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, the commands and responses are always sent between the host computer and one selected controller.

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 device number (1 to 16) of Mercury™ controllers using the address DIP switches on the front panel. The device number determines the default identifiers of the corresponding axes, I/O channels and joystick connections.

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".

The default identifiers can be changed using Mercury_SAI() (p. 45). The new identifiers must then be used with all szAxes arguments and in macro names, even for macros that were previously stored using different names.

1.3.2. I/O Line Designators

Each Mercury™ and Mercury™ Step controller provides four digital input and four digital output lines on the "I/O" socket. 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. Note that when the digital output line 4 of a Mercury™ controller is used with Mercury_CTO() to trigger other devices (pin 8 of the "I/O" socket, see User manual for pinout), its ID corresponds with the device number of the controller to which it belongs.

The four digital input lines of Mercury™ and Mercury™ Step controllers can also be used for analog input (0 to 5 V). In regard to analog input, these input channels have IDs from A1 to A64, again depending on the controller's address settings, and skipping values associated with any missing addresses.

Example: A network consists of a C-863 DC Motor Controller with device number 1 (DIP switches 1 to 4 are all ON) and a C-663 Stepper Controller with device number 3 (DIP switches 1 to 4 are set ON ON OFF ON). The GCS DLL will provide:

- Axes "A" and "C"
- Digital I/O using channel IDs A, B, C, D and I, J, K, L
- Analog input using channel IDs A1, A2, A3, A4 and A9, A10, A11, A12
- Trigger output channels: 1 and 3

1.3.3. Controller Joystick Connections

C-863 and C-663 Mercury™ controllers can be connected to an analog joystick device using their "Joystick" socket (see controller User manual for pinout). For that purpose, PI provides C-819.20 2-axis or C-819.30 3-axis joystick models. C-819.20 2-axis joystick devices have only one connection cable. Therefore an Y-cable (C-819.20Y) must be used to connect one axis and one logical button of the joystick to one controller and the other axis and other button to another controller. C-819.30 3-axis joystick devices are already equipped with separate connection cables for 3 different controllers. Since Mercury™ controllers support only one joystick axis and button, the corresponding identifiers to be used in GCS commands are always 1. The distinction between the individual axes and buttons is in fact made by the ID of the joystick device. The joystick device ID is identical to the device ID of the controller to which the joystick is connected (set with the DIP switches 1 to 4, can be 1 to 16).

Example: A network consists of a C-863 DC Motor Controller with device number 1 (DIP switches 1 to 4 are all ON) and a C-663 Stepper Controller with device number 3 (DIP switches 1 to 4 are set ON ON OFF ON). In GCS commands the identifiers to be used then are as follows:

- Joystick devices: 1 and 3
- Joystick axes: 1 and 1
- Joystick buttons: 1 and 1

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 PI Mercury GCS library.

- Quick Start (p. 9) enables you to operate your system and gives sample code.
- DLL Handling (p. 9) explains how to load the library and how to access the functions provided by the MERCURY DLL.
- Function Calls (p. 14) and Types Used in PI Software (p. 15) provides some general information about the syntax of most commands in the DLL.
- Native Command Gateway (p. 16) shows how to initiate communication with a Mercury™ Class controller or controller network (see also Interface Settings (p. 22)).
- Mercury™ Commands (p. 23) describes the functions encapsulating the embedded GCS commands for Mercury™ Class controllers
- Macro Storage on Controller (p. 48) explains the macro storage facility and the mechanism used by the PI Mercury GCS DLL for macro creation.
- Motion Parameters Overview (p. 55) describes how to handle the stage parameters and list the valid parameter set.
- Error Code (p. 64) has a description of the possible errors.

2. Quick Start

2.1. Software Installation

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

Windows operating systems:

1 Insert the product CD in your host PC.

2 If the Setup Wizard does not open automatically, start it from the root directory of the CD with the  icon.

3 Follow the on-screen instructions and select the “typical” installation. Typical components are GCS LabView drivers, Native and GCS DLLs, PIMikroMove™, MMCRun and all manuals..

Linux operating systems:

1 Insert the product CD in the host PC.

2 Open a terminal and go to the /linux directory on the CD.

3 Log in as superuser (root).

4 Start the install script with ./INSTALL
Keep in mind the case sensitivity of Linux when typing the command.

5 Follow the on-screen instructions. You can choose the individual components to install.

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

NOTE

The PIStages2.dat stage database file needed by the PI Mercury GCS DLL is installed in the ...\PI\GcsTranslator directory. In that directory, also the MercuryUserStages2.dat database will be located which is created automatically the first time you connect stages in the host software (i.e. the first time Mercury_qVST() or Mercury_CST() are called).
The location of the PI directory is that specified upon installation, by default C:\Documents and Settings\All Users\Application Data (Windows XP) or C:\ProgramData (Windows Vista). If this directory does not exist, the EXE file that needs the stage databases will look in its own directory.

See Sections 3, 4 and 5 for more information about PI DLLs.

The PI host software is improved continually. It is therefore recommended that you visit the PI website (www.pi.ws) regularly to see if updated releases of the software are available for download. Updates are accompanied by information (readme files) so that you can decide if updating makes sense for your application. You need a password to see if updates are available and to download them. This password is provided on the product CD in the *Produktname*Releasenews PDF file in the \Manuals directory. See "Software Updates" in the User Manual of your controller for download details.

2.2. Connect the Controller

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

2.3. Install USB Drivers

When using the USB interface for the first time, two FTDI USB drivers must be installed on the host PC. These drivers are provided on the Mercury™ CD in the \USB_Driver directory. Follow the on-screen instructions. Installing the USB drivers requires administrator rights on the host PC.

NOTE

A USB connection will appear as an extra COM port when the controller is connected, powered up, and the USB drivers are installed.

The baud rate used by the host must be the same as that set on the DIP switches, even if the USB interface is used!

2.4. Starting Up

NOTE

When you are working with the host software from PI (e.g. the PI Mercury GCS DLL or PIMikroMove™), you simply select the suitable stage parameter set from a stage database (e.g. by calling Mercury_CST()) to adapt the Mercury™ controller to the connected stage. The stage selection from the database must be repeated whenever you replace the connected stage with one of another stage type. See "Motion Parameters Overview" on p. 55 for further information.

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

1. Open a connection between the host PC and the Mercury™ network by calling Mercury_ConnectRS232(). See "Communication Initialization" on p. 20 for details.
2. Call Mercury_CST() to determine which stage is connected to the Mercury™ controller. Mercury_CST() loads the specific values for the connected stage from a stage database (see also "Stage Definition" on p. 17) and sends them to the controller so that the controller parameters (see p. 55) are properly adjusted to the connected mechanics.
3. Call Mercury_INI() to initialize the stage and switch the servo on. The stage must be referenced before you can make absolute moves with functions like Mercury_MOV(). By default, referencing must be done using Mercury_REF(), Mercury_MNL() and Mercury_MPL(), depending on the connected mechanics. Use Mercury_IsControllerReady() in a loop to observe completion of the referencing procedure, see also Section 2.5.
4. Make a few test moves with Mercury_MOV() so that you can verify your program's operation.

2.5. Referencing

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

- The axis can be referenced. This involves moving it until it 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.

2.6. Sample

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

The following example shows how to connect to a Mercury™, and (without the call printf()) represents a typical initialization.

```
//ID = Mercury_InterfaceSetupDlg("PI\\MercuryDLLSample");
ID = Mercury_ConnectRS232(1, 9600);
if (ID < 0 || Mercury_IsConnected(ID)==FALSE)
{
        printf("Could not connect to Mercury\n");
        return false;
}
char szIDN[1000];
if (!Mercury_qIDN(ID, szIDN, 999))
{
        printf("qIDN() failed!\n");
        return false;
        }
printf("ID of PI Mercury = \"%s\"\n\n", szIDN);

char szAxes[18];
if (!Mercury_qSAI_ALL(ID, szAxes, 17))      {
        return false;
}

size_t nrAxes = strlen(szAxes);
char szStages[1000] = "M-122.2DD\nM-112.12S\n";
if (!Mercury_CST(ID, szAxes, szStages) ) return false;

if (!Mercury_qCST(ID, szAxes, szStages, 999) ) return false;
printf("connected stages: \n\"%s\"\n", szStages);

if (!Mercury_INI(ID, szAxes) ) return false;

printf("Start move to negative limit switch for axes %s\n", szAxes);
if (!Mercury_MNL(ID, szAxes) ) return false;
BOOL bReferencing = FALSE;
do
{
        if (!Mercury_IsReferencing(ID, NULL, &bReferencing)) return false;
        Sleep(500);
        printf(".");
} while (bReferencing);
printf("\n");

BOOL refOK[16];
if (!Mercury_IsReferenceOK(ID, szAxes, refOK) ) return false;
int nrNotOK = 0;
for (i=0; i<nrAxes; i++)
{
        printf("axis %c: reference %s\n", szAxes[i], refOK[i] ? "OK" : "not OK");
        if (refOK[i] != 1)
                nrNotOK++;
```

```
        }
        if (nrNotOK >0)
        {
                printf("some axes not referenced!\n");
                return false;
        }

        double negRangeLimit[16];
        double posRangeLimit[16];
        double pos[16];
        if (!Mercury_qTMN(ID, szAxes, negRangeLimit) ) return false;
        if (!Mercury_qTMX(ID, szAxes, posRangeLimit) ) return false;
        if (!Mercury_qPOS(ID, szAxes, pos) ) return false;
        double target[16];
        for (i=0; i<nrAxes; i++)
        {
                target[i] = ((std::min)(posRangeLimit[i],10000.0) + (std::max)(negRangeLimit[i], -
10000.0))/2;
                printf("axis %c: Range %g - %g, current position: %g, move to %g\n",
                        szAxes[i], negRangeLimit[i], posRangeLimit[i], pos[i], target[i]);
        }

        if (Mercury_MOV(ID, szAxes, target)!=1) return false;
        BOOL bMoving = FALSE;
        do
        {
                if (!Mercury_IsMoving(ID, NULL, &bMoving)) return false;
                Sleep(500);
                printf(".");
        } while (bMoving);
        printf("\n");

        if (!Mercury_qPOS(ID, szAxes, pos) ) return false;
        for (i=0; i<nrAxes; i++)
        {
                printf("axis %c: current position: %g\n",
                        szAxes[i], pos[i]);
        }
```

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. 15)). If the function succeeded, the return value is **TRUE**, otherwise it is **FALSE**. To find out what went wrong, call **Mercury_GetError**()(p. 20)) and look up the value returned in “Error Code” (p. 64). 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. 32) and **Mercury_qPOS**() (p. 39))

See “Types Used in PI Software” (p. 15) 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. 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

7. Stage Definition

7.1. Stage Database Files

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

New (user-defined) stages are all stored in *MercuryUserStages2.dat* and known PI stages are in *PiStages2.dat. PiStages2.dat* may not be edited, but updated versions are made available regularly from PI, see “Updating PIStages2.dat” on p. 19 for details.

If an older version of the software was installed an existing MercuryUserStages.dat is automatically converted into MercuryUserStages2.dat.

For parameter descriptions see the “Parameter List” Section (p. 55).

NOTE

The PIStages2.dat stage database file needed by the PI Mercury GCS DLL is installed in the ...\PI\GcsTranslator directory. In that directory, also the MercuryUserStages2.dat database will be located which is created automatically the first time you connect stages in the host software (i.e. the first time Mercury_qVST() or Mercury_CST() are called).
The location of the PI directory is that specified upon installation, by default C:\Documents and Settings\All Users\Application Data (Windows XP) or C:\ProgramData (Windows Vista). If this directory does not exist, the EXE file that needs the stage databases will look in its own directory.

7.2. How to Define Stage Parameter Sets

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)

To create a valid parameter set for a new stage, you can use the Mercury_SPA function call (p. 45). See the parameter handling description starting on p. 55 for further details. Note that the parameter which determines whether a stage is “new” or not is the *Name* parameter (ID 0x3C). 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 Mercury_INI().

You can ease the creation by loading an existing parameter set with Mercury_CST() (p.26) and afterwards change the name and any other parameters, which differ, with Mercury_SPA(). (Mercury_CST() “connects” a valid stage, i.e. makes its parameter set active. It uses the corresponding parameters in the stage database 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 Mercury_CST(), use Mercury_AddStage() (p.) to add its parameter set to *MercuryUserStages2.dat*. After addition to *MERCURYUserStages2.dat* the stage will also appear in the list returned by Mercury_qVST() (p.).

If you want to remove a stage from *MercuryUserStages2.dat* call Mercury_RemoveStage() (p.18).

It may be more comfortable to set the stage parameters using the *PIStageEditor* (a GUI dialog). See the separate PI Stage Editor manual (SM144E) for a description of how to operate that graphic interface.

The *PIStageEditor* can also be started from *PIMikroMove*™. This program provides several functions which ease creating and editing stage parameter sets. For further information, refer to “Tutorials - Frequently Asked Questions” in the *PIMikroMove*™ manual.

NOTES

The Mercury_OpenUserStagesEditDialog() or Mercury_OpenPiStagesEditDialog() functions are provided for compatibility reasons only and should not be used to open the *PIStageEditor*. Since the *PIStageEditor* is not modal, problems can occur when the calling application exits before the *PIStageEditor* window is closed. Please start the *PIStageEditor* either from *PIMikroMove*™ or via its executable.

The GCS DLL only accepts the DAT-files *PiStages2.dat* and *MercuryUserStages2.dat*. Although it is possible to save DAT-files with any user-defined names, they are not used by the software.

7.3. Documentation of Stage Definition Functions

When defining user stages, it is important to set the stage parameters correctly. See the Mercury_qSPA function call on p. 40 for the parameters most frequently accessed by the user, for a complete list see the “Motion Parameters Overview” Section (p. 11).

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

CAUTION: This function is provided for compatibility reasons only. It is not recommended to open the *PIStageEditor* this way. Since the *PIStageEditor* is not modal, problems can occur when the calling application exits before the *PIStageEditor* window is closed. Please start the *PIStageEditor* either from *PIMikroMove*™ or via its executable to check stage parameter sets in *PiStages2.dat*.

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.

CAUTION: This function is provided for compatibility reasons only. It is not recommended to open the *PIStageEditor* this way. Since the *PIStageEditor* is not modal, problems can occur when the calling application exits before the *PIStageEditor* window is closed. Please start the *PIStageEditor* either from *PIMikroMove*™ or via its executable to edit, add or remove stage parameter sets in *MercuryUserStages2.dat*..

Arguments:

iId ID of controller

Returns:

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

7.4. Updating PIStages2.dat

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

1 On the www.pi.ws front page, click on *Download/Support* in the *Service* section on the left

2 On the *Download/Support* page, click on *Manuals and Software*

3 Click on *Download* in the navigation bar across the top (no login or password is required)

4 Click on the *General Software* category

5 Click on *PI Stages*

6 Click the download button below *PIStages2.dat*

7 In the download window, switch to the ...\PI\GcsTranslator directory. The location of the PI directory is that specified upon installation, by default C:\Documents and Settings\All Users\Application Data (Windows XP) or C:\ProgramData (Windows Vista) (may differ in other-language Windows versions).
Hint: You can identify the path using the *Version Info…* item in the controller menu in PIMikroMove™.

8 If desired, rename the existing PIStages2.dat (if present) so as to preserve a copy for safety reasons

9 Download the file from the server as PIStages2.dat

8. Communication Initialization

8.1. 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)

8.2. 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.

8.3. 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. 35) 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. 64) 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. 22) 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. 20) 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. 20).
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

8.4. Interface Settings

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

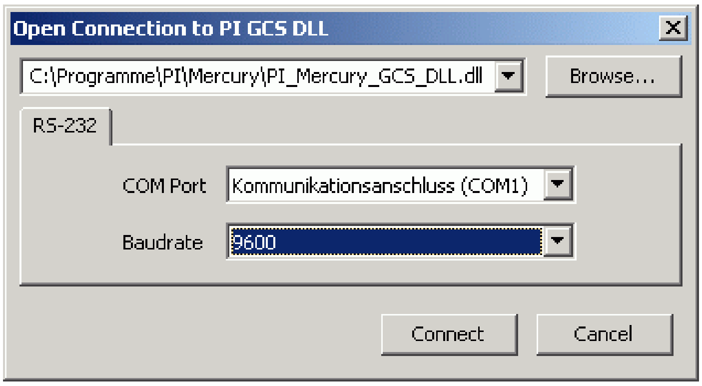
NOTE

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, with the USB drivers installed on the host PC. 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.

- `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™ 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.



9. Mercury™ Commands

9.1. Function Overview

Function	Short Description	Page
BOOL **Mercury_MNL** (int ID, const char* szAxes)	reference axes to negative Lim	32
BOOL **Mercury_MOV** (int ID, const char* szAxes, double *pdValarray)	move to given absolute position	32
BOOL **Mercury_MPL** (int ID, const char* szAxes)	reference axes to positive Lim	33
BOOL **Mercury_MVR** (int ID, const char* szAxes, double *pdValarray	move relatively by given distance	33
BOOL **Mercury_POS** (int ID, const char* szAxes, double *pdValarray)	set actual position	33
BOOL **Mercury_qBRA** (int ID, char *axes, int maxlen)	get axes with brakes	34
BOOL **Mercury_qCST** (int ID, const char* szAxes, char *names, int maxlen)	query stage assignment to axes	34
BOOL **Mercury_qCTO** (int ID, const long* piTriggerOutputIdsArray, const long* piTriggerParameterArray, char* szBuffer, int iArraySize, int iBufferMaxlen)	query trigger parameter	34
BOOL **Mercury_qDFF (**int ID, const char* szAxes, double * pdValarray)	query factor	35
BOOL **Mercury_qDFH** (int ID, const char* szAxes, double *pdValarray)	query home position	35
BOOL **Mercury_qDIO** (int ID, const char* szChannels, BOOL *pbValarray)	query digital input lines	35
BOOL **Mercury_qERR** (int ID, int *pError)	query controller error	35
BOOL **Mercury_qHLP** (int ID, char *buffer, int maxlen)	display this help message	36
BOOL **Mercury_qHPA** (int *ID*, char* *szBuffer*, int *iBufferSize*)	display parameter help message	36
BOOL **Mercury_qIDN** (int ID, char *buffer, int maxlen)	query identification string of controller	36
BOOL **Mercury_qJAS** (int ID, const int* iJoystickIDsArray, const int* iAxesIDsArray, double* pdValarray, int iArraySize)	get joystick value	36
BOOL **Mercury_qJAX** (int ID, const int* iJoystickIDs, const int* iAxesIDs, int iArraySize, char* szAxesBuffer, int iBufferSize)	query joystick axis	37
BOOL **Mercury_qJBS** (int ID, const int* iJoystickIDsArray, const int* iButtonIDsArray, BOOL* pbValarray, int iArraySize)	query if joystick button pressed	37
BOOL **Mercury_qJLT**(int ID, const int* iJoystickIDsArray, const int* iAxisIDsArray, int iNumberOfTables, int iOffsetOfFirstPointInTable, int iNumberOfValues, double** pdValueArray, char* szGcsArrayHeader, int iGcsArrayHeaderMaxSize)	Query active joystick table	38
BOOL **Mercury_qJON** (int ID, const int* iJoystickIDs, BOOL* pbValarray, int iArraySize)	query if joystick is enabled	38
BOOL **Mercury_qLIM** (int ID, const char* szAxes, BOOL *pbValarray)	query presence of limit switches	38
BOOL **Mercury_qMOV** (int ID, const char* szAxes, double *pdValarray)	query target position	39
BOOL **Mercury_qONT** (int ID, const char* szAxes, BOOL *pbValarray)	query whether stage is on target	39

9.2. Function Documentation

These functions encapsulate the GCS ASCII commands supported by Mercury™ controllers and provide some shortcuts to make the work with these controllers easier. See “Function Calls“ (p. 14) for some general notes about the parameter syntax. “Types Used in PI Software“ (p. 15) 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™ controller and the DLL addresses commands for that axis to that controller.

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*. Is ignored by Mercury™ controllers, provided only for compatibility reasons.

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_CTO** (int *ID*, const int* *piTriggerOutputIdsArray*, const int* *piTriggerParameterArray*, const double* *pdValueArray*, int *iArraySize*)

Corresponding command: `CTO`

Configures the trigger output conditions for the given trigger output line. On Mercury™ controllers, the digital output line 4 can be configured for trigger output (pin 8 of the “I/O” socket, see User manual for pinout).

The trigger output conditions will become active when activated with Mercury_TRO(). Do not use Mercury_DIO() on any physical trigger line for which trigger output is enabled with Mercury_TRO().

Arguments:

ID ID of controller network
piTriggerOutputIdsArray is an array with the trigger output lines of the Mercury™ network. Note that the ID of a trigger output line corresponds with the device number of the controller to which it belongs (the controller device number is set with DIP switches 1 to 4 on the controller front panel).
piTriggerParameterArray is an array with the CTO parameter IDs:
- 1 = TriggerStep
- 2 = Axis
- 3 = TriggerMode
- 10 = TriggerPosition

pdValueArray is an array of the values to which the CTO parameters are set:
- for TriggerStep: step size in physical units
- for Axis: the axis to connect to the trigger output line. The assignment is fixed (the axis identifier must correspond to the controller address, see “Axis Designators” on p. 7 for details).
- for TriggerMode:
  - 0 = PositionDistance: A trigger pulse is written whenever the axis has covered the distance given by TriggerStep.
  - 7 = Position + Offset: The first trigger pulse is written when the axis has reached the trigger position given by TriggerPosition. The next trigger pulses each are written when the axis position equals the sum of the last valid trigger position and the increment value given by TriggerStep.
  - 8 = SingleTrigger: A trigger pulse is written when the axis has reached the trigger position given by TriggerPosition.
- for TriggerPosition: position where a trigger pulse is to be output, in physical units

iArraySize is the size of the buffer *pdValueArray*

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. 60.

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.

Note: Do not use Mercury_DIO() on any physical trigger line for which trigger output is enabled with Mercury_TRO().

Parameters:

ID ID of controller network
szChannels string with digital output channel identifiers (p. 7); 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. 28).

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. 28)).

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. 44), **Mercury_MNL**() (p. 32) or **Mercury_MPL**() (p. 33). If still referencing or no reference move was started since startup of library, the result is 0. Call **Mercury_qREF**() (p. 39) to see which axes have a reference switch. **Mercury_REF**() can be used only for axes with reference switches, **Mercury_MNL**() (p. 32) and **Mercury_MPL**() (p. 33) 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_STP(**), p. 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, switches servo on, 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*)

Corresponding command: #5

Check if *szAxes* are moving. If an axis is moving, the corresponding element of the array will be **TRUE**, otherwise **FALSE.** If no axes 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_IsReferenceOK** (int ID, const char* *szAxes*, BOOL * *pbValarray*)

Check the reference state of the given axes. Call **Mercury_qREF**() (p. 39) to find out which axes have a reference switch. Axes with a reference switch can be referenced with **Mercury_REF**() (p. 44); axes with limit switches with **Mercury_MNL**() (p. 32) or **Mercury_MPL**() (p. 33).

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_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.
To ensure that the loaded response table is used, call Mercury_JDT() before you enable joystick operation with Mercury_JON().

Arguments:

ID ID of controller network
iJoystickIDs device numbers of the Mercury™ controllers (the joystick ID is identical to the device number of the controller to which the joystick is connected; set with the DIP switches 1 to 4 on the controller front panel, can be 1 to 16)
piValarray pointer to array with table types for the corresponding joystick axes
iArraySize size of arrays

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: JLT

Fill the joystick response table with values.
The amplitudes of the joystick axes (i.e. their displacements) are mapped to the current valid velocity settings of the controller axes. For each joystick axis there is a response table that defines this mapping. With Mercury_JLT() this table can be written (a default table provided by the controller can be loaded with Mercury_JDT()).
To ensure that the loaded response table is used, call Mercury_JLT() before you enable joystick operation with Mercury_JON().

Arguments:

ID ID of controller network
iJoystickID device numbers of the Mercury™ controllers (the joystick ID is identical to the device number of the controller to which the joystick is connected; set with the DIP switches 1 to 4 on the controller front panel, can be 1 to 16)
iAxisID must be 1 for Mercury
iStartAdress start point of the response table, starts with 1
pdValueArray pointer to array with values for the joystick table
iArraySize number of values to be written

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 joystick device (i.e. for the Mercury™ controller to which the joystick is connected). To enable, set the corresponding entry in pbValarray to TRUE..

Do not enable controllers with no physical joystick connected, as uncontrolled motion could occur.

Arguments:

ID ID of controller network
iJoystickIDs device numbers of the Mercury™ controllers (the joystick ID is identical to the device number of the controller to which the joystick is connected; set with the DIP switches 1 to 4 on the controller front panel, can be 1 to 16)
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_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. 31) to find out if the axes are still moving and **Mercury_GetRefResult**() (p. 29) 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. 46) 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. 31) to find out if the axes are still moving and **Mercury_GetRefResult**() (p. 29) 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. 28) 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, intmaxlen)

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 space character (except of the last line) and line-feed character. Example

A=M-227.10 SP LF

C=DEFAULT_STAGE-S LF

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_qCTO** (int ID, const long* piTriggerOutputIdsArray, const long* piTriggerParameterArray, char* szBuffer, int iArraySize, int iBufferMaxlen)

Corresponding command: CTO?

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

Arguments:

ID ID of controller network
piTriggerOutputIdsArray is an array with the trigger output lines of the Mercury™ network. Note that the ID of a trigger output line corresponds with the device number of the controller to which it belongs (the controller device number is set with DIP switches 1 to 4 on the controller front panel).
piTriggerParameterArray is an array with the CTO parameter ID's
szBuffer buffer to receive the values to which the CTO parameters are set, see Mercury_CTO() for details
iArraySize is the size of the buffer *piTriggerOutputIdsArray*
iBufferMaxlen size of *szBuffer*

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. 20) 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_qHPA** (int *ID*, char* *szBuffer*, int *iBufferSize*)

Corresponding **command:** `HPA?`

Lists a help string which contains all parameters provided by the PI Mercury GCS DLL with short descriptions. See “Motion Parameters Overview” beginning on p. 55 for parameter handling and for an appropriate list of all parameters available for Mercury™ controllers.

Arguments:

ID ID of controller network
szBuffer buffer to receive the string read in from the PI Mercury GCS DLL, lines are separated by '\n' ("line-feed")
iBufferSize size of *szBuffer*, must be given to avoid buffer overflow.

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: `*IDN?`

Get identification string of the PI Mercury GCS DLL.

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_qJAS** (int ID, const int* iJoystickIDsArray, const int* iAxesIDsArray, double* pdValarray, int iArraySize)

Corresponding command: `JAS?`

Get the current status of the given axis of the given joystick device which is directly connected to the controller. The reported factor is applied to the velocity set with Mercury_VEL(), the range is -1.0 to 1.0.

Arguments:

ID ID of controller network
iJoystickIDsArray array with device numbers of the Mercury™ controllers (the joystick ID is identical to the device number of the controller to which the joystick is connected; set with the DIP switches 1 to 4 on the controller front panel, can be 1 to 16)
iAxesIDsArray array with axis IDs of the joystick axes (must be 1 for Mercury™ controllers, which only have 1 joystick axis per device)

pdValarray pointer to array to receive the joystick axis amplitude, i.e. the factor which is currently applied to the current valid velocity setting of the controlled motion axis; corresponds to the current displacement of the joystick axis.
iArraySize size of arrays

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **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 device numbers of the Mercury™ controllers (the joystick ID is identical to the device number of the controller to which the joystick is connected; set with the DIP switches 1 to 4 on the controller front panel, can be 1 to 16)
iAxesIDs array with axis IDs of the joystick axes (must be 1 for Mercury™ controllers, which only have 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_qJBS** (int ID, const int* iJoystickIDsArray, const int* iButtonIDsArray, BOOL* pbValarray, int iArraySize)

Corresponding command: JBS?

Get the current status of the given button of the given joystick device which is directly connected to the controller.

Arguments:

ID ID of controller network
iJoystickIDsArray array with device numbers of the Mercury™ controllers (the joystick ID is identical to the device number of the controller to which the joystick is connected; set with the DIP switches 1 to 4 on the controller front panel, can be 1 to 16)
iButtonIDsArray array with IDs of the joystick buttons (must be 1 for Mercury™ controllers, which only support 1 joystick button per device)
pbValarray pointer to array to receive the joystick button state, indicates if the joystick button is pressed; 0 = not pressed, 1 = pressed
iArraySize size of arrays

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qJLT**(int ID, const int* iJoystickIDsArray, const int* iAxisIDsArray, int iNumberOfTables, int iOffsetOfFirstPointInTable, int iNumberOfValues, double** pdValueArray, char* szGcsArrayHeader, int iGcsArrayHeaderMaxSize)

Corresponding command: `JLT?`

Reading of the current valid response table values. Detailed information about the data read in can be found in the header sent by the controller. See the GCS Array manual (SM146E) for details.

Arguments:

ID ID of controller network
iJoystickIDsArray array with device numbers of the Mercury™ controllers (the joystick ID is identical to the device number of the controller to which the joystick is connected; set with the DIP switches 1 to 4 on the controller front panel, can be 1 to 16)
iAxisIDsArray IDs of axes; must all be 1 for Mercury
iNumberOfTables number of record tables to read
iOffsetOfFirstPointInTable index of first value to be read (starts with index 1)
iNumberOfValues number of values to read
pdValueArray pointer to internal array to store the data; data from all tables read will be placed in the same array with the values interspersed; the DLL will allocate enough memory to store all data, call Mercury_GetAsyncBufferIndex() to find out how many data points have already been transferred
szGcsArrayHeader buffer to store the GCS array header
iGcsArrayHeaderMaxSize size of the buffer to store the GCS array header, must be given to prevent buffer overflow

Returns:

TRUE if successful, **FALSE** otherwise

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

Corresponding command: `JON?`

Gets joystick enable/disable states for given joystick devices (i.e. for the Mercury™ controllers to which the joysticks are connected).

Arguments:

ID ID of controller network
iJoystickIDs device numbers of the Mercury™ controllers (the joystick ID is identical to the device number of the controller to which the joystick is connected; set with the DIP switches 1 to 4 on the controller front panel, can be 1 to 16)
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_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. 55 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 specified 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 5 for the C-863):

C-863 DC-motor versions:

Bit 0: not used
Bit 1: Reference signal (input)
Bit 2: Positive limit signal (input)
Bit 3: Negative limit signal (input)
Bit 4: DIO 1
Bit 5: DIO 2
Bit 6: DIO 3
Bit 7: DIO 4

C-663 stepper motor versions:

Bit 0: Limit negative
Bit 1: Reference signal
Bit 2: Limit positive
Bit 3: no function
Bit 4: Digital input 1
Bit 5: Digital input 2
Bit 6: Digital input 3
Bit 7: Digital input 4

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: SVO?

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. With Mercury™ controllers, the response contains only the analog channels on the I/O connector.

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 Mercury™ controller. This is the maximum possible number of joystick axes that can be connected to the network.

Parameters:

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

Returns:

TRUE if successful, **FALSE** otherwise

BOOL **Mercury_qTRO** (int ID, const int* *piTrigger lines*, BOOL * *pbValarray, int iArraySize*)

Corresponding command: `TRO?`

Gets trigger output-mode enable-status for given trigger output line (the trigger output configuration is made with Mercury_CTO()).

Arguments:

ID ID of controller network
piTrigger lines is an array with the trigger output lines of the Mercury™ network. Note that the ID of a trigger output line corresponds with the device number of the controller to which it belongs (the controller device number is set with DIP switches 1 to 4 on the controller front panel).
pbValarray pointer to array to receive modes of the specified trigger lines: **TRUE** for "enabled", **FALSE** for "disabled"
iArraySize number of trigger lines

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 versions of the controller firmware and of the PI Mercury GCS DLL.

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 the PI Mercury GCS DLL (the content of the stage PiStages2.dat and MercuryUserStages2.dat database files), 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. 31) to find out if the axes are still moving and **Mercury_GetRefResult**() (p. 29) 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. 46) 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. 43). 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*. See Section 11 on p. 55 for a list of valid parameter IDs.

Mercury_SPA has two arrays as arguments. The first array has the parameters which have to be modified, the second one the values.

Example (parameter IDs are given in decimal format): If you want to set the velocity (ID=10) to 0.05, the acceleration (ID=11) to 8 and the scaling factor (ID=18) to 25.4 (converts the physical unit of mm to inches), you can use the following code (in C(++) syntax):

```
char szAxes[] = “AAA”;
int cmd[] = {10, 11, 18};
double values[] = {0.05, 8, 25.4};
Mercury_SPA(id, szAxes, cmd, values);
```

szAxes = "AAA"	cmd = {10, 11, 18}	values = {0.05, 8, 25.4}
szAxes[0] = 'A'	cmd[0] = 10	values[0] = 0.05
szAxes[1] = 'A'	cmd[1] = 11	values[1] = 8

`szAxes[2] = 'A'`	`cmd[2] = 18`	`values[2] = 25.4`

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, 11}, {0.06, 0.05, 9})` will set the velocity of 'A' to 0.05 and the acceleration 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_TRO** (int ID, const int* *piTrigger lines*, BOOL * *pbValarray*, int *iArraySize*)

Corresponding command: TRO

On Mercury™ controllers, the digital output line 4 can be configured for trigger output (pin 8 of the “I/O” socket, see User manual for pinout). Mercury_TRO() enables or disables the **TR**igger **O**utput mode which was set with Mercury_CTO(). If *pbValarray[index]* is **FALSE** the mode is "off", if **TRUE** it is set to "on".

Do not use Mercury_DIO() on any physical trigger line for which trigger output is enabled with Mercury_TRO().

Arguments:

ID ID of controller
piTrigger lines is an array with the trigger output lines of the Mercury™ network. Note that the ID of a trigger output line corresponds with the device number of the controller to which it belongs (the controller device number is set with DIP switches 1 to 4 on the controller front panel).
pbValarray pointer to boolean array with modes for the specified trigger lines, **TRUE** for "on", **FALSE** for "off"
iArraySize number of trigger lines

Returns:

TRUE if successful, **FALSE** otherwise

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

10. Macro Storage on Controller

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

10.1. 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-on 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.

10.2. 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.

10.3. Macro Translation by the GCS DLL

10.3.1. 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: aMC0nn 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)
    - 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()

---

* 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), Access to the following SPA parameters by macros is permitted: all others will be ignored (parameter IDs in hexadecimal / decimal format):

- 0x1 / 1: P-Term
- 0x2 / 2: I-Term
- 0x3 / 3: D-Term
- 0x4 / 4: I-Limit
- 0x8 / 8: Max.Position Error
- 0xA / 10: Max. Velocity
- 0xB / 11: Max Acceleration (muss >200 sein)
- 0x18 / 24: Limit Switch Mode
- 0x32 / 50: No Limit Switch
- 0x40 / 64: Hold Current (HC native command) in mA
- 0x41 / 65: Drive Current (DC native command) in mA
- 0x42 / 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

10.3.2. 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 MDn, TMn, 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>”

10.3.3. 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.

10.4. Macro Function Overview

Function	Short Description	Page
BOOL **Mercury_DEL** (int ID, double dSeconds)	Delays execution of macro (only in macro)	52
BOOL **Mercury_IsRecordingMacro** (int ID, BOOL *pbRecordingMacro)	Check if controller is currently recording a macro.	52
BOOL **Mercury_IsRunningMacro** (int ID, BOOL *pbRunningMacro	Check if controller is currently running a macro	52
BOOL **Mercury_MAC_BEG** (int ID, const char *szName)	Put the DLL in macro recording mode.	52
BOOL **Mercury_MAC_DEL** (int ID, const char *szName)	Delete macro with name szName.	53
BOOL **Mercury_MAC_END** (int ID)	Take the DLL out of macro recording mode.	53
BOOL **Mercury_MAC_NSTART** (int ID, const char *szName, int nrRuns)	Start macro with name szName. The macro is repeated nrRuns times.	53
BOOL **Mercury_MAC_START** (int ID, const char *szName)	Start macro with name szName.	53
BOOL **Mercury_MEX** (int ID, const char *szCondition)	Stop Macro EXecution due to a given condition	54
BOOL **Mercury_qMAC** (int ID, char *szName, char *szBuffer, int maxlen)	Get available macros, or list contents of a specific macro.	54
BOOL **Mercury_WAC** (int ID, const char *szCondition)	WAit until a given Condition occurs	54

10.5. Macro Function Documentation

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

Corresponding command: `DEL`

Delays execution of macro (only in macro)

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_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 Translation by the GCS DLL”, beginning on p. 48 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_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_MAC_BEG** (int ID, char * *szName*)

Corresponding command: `MAC BEG`

Put the DLL in macro recording mode. See “Macro Translation by the GCS DLL,” beginning on p. 48 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. 54). See "Macro Translation by the GCS DLL," beginning on p. 48 for 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 Translation by the GCS DLL," beginning on p. 48 for 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. 54). See "Macro Translation by the GCS DLL," beginning on p. 48 for 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. 54). See "Macro Translation by the GCS DLL," beginning on p. 48 for 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 Translation by the GCS DLL," beginning on p. 48 for further details.)

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_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_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.

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

The hardware basics of the connected stage and—for C-863 DC motor controllers— the required closed-loop control settings are mirrored in parameters of the PI Mercury GCS DLL. The parameter values have to be adjusted properly before initial operation of a stage.

With Mercury_qHPA() you can obtain a list of all available parameters with information about each (e.g. short descriptions). The current parameter values can be read with the Mercury_qSPA() function.

Using the "general" modification function Mercury_SPA() parameters can be changed in volatile memory. In addition to the "general" modification function, there are some functions which change certain specific parameters (see table below).

CAUTION

Wrong values of the parameters may lead to improper operation or damage of your hardware. Be careful when changing parameters.

The interrelation of the hardware-dependent parameters 0x15, 0x16, 0x17, 0x2F and 0x30 is described in "Travel Range Adjustment” on p. 61.

C-863 DC motor controllers only: Find details regarding closed-loop control in “Control Options” in the C-863 User manual.

To store parameter values, save them to the MERCURYUserStages2.dat stage database (see “Stage Definition” on p. 17) or create an Autostart macro which sets the parameter values after each power-on or reset of the controller (see “Macro Storage on Controller” on p. 48 and the PIMikroMove™ manual for details).

11.2. Parameter List

Parameter lines with white background apply to all controller versions, lines with light gray background apply to C-663 stepper motor versions only, and lines with dark gray background apply to C-863 DC motor versions only.

Parameter ID (hexadecimal / decimal)	Data Type	Parameter Description	Possible Values/Notes
0x1 / 1	FLOAT	P-term for position control	0 to 32767
0x2 / 2	FLOAT	I-term for position control	0 to 32767
0x3 / 3	FLOAT	D-term for position control	0 to 32767
0x4 / 4	FLOAT	I-limit for position control	0 to 32767
0x8 / 8	FLOAT	Maximum position error (user unit)	0 to 32767 Used for stall detection. If the position error (i.e. the absolute value of the difference between current position and commanded position) in closed-loop operation exceeds the given maximum, the PI Mercury GCS DLL sets error code -1024 ("Motion error"), the servo will be switched off and the axis stops.
0xA / 10	FLOAT	Maximum closed-loop velocity (user unit/s)	> 0 Gives the maximum value for parameter 0x49.
0xB / 11	FLOAT	Current closed-loop acceleration (user unit/s^2)	Gives the current acceleration, limited by parameter 0x4A
0xE / 14	FLOAT	Numerator of the counts-per-physical-unit factor	1 to 2147483647 for each parameter. The counts-per-physical-unit factor determines the "user" unit for closed-loop motion commands. When you change this factor, all other parameters whose unit is based on the "user" unit must be adapted too, e.g. closed-loop velocity and parameters regarding the travel range. Note: To customize your physical unit use parameter 0x12 instead.
0xF / 15	FLOAT	Denominator of the counts-per-physical-unit factor	
0x11 / 17	FLOAT	Invert direction	-1 to invert the direction, else 1
0x12 / 18	FLOAT	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 Mercury_DFF() to change this factor.
0x13 / 19	FLOAT	Rotary stage	1 = rotary stage, else 0
0x14 / 20	FLOAT	Stage has a reference	1 = the stage has a reference, else 0
0x15 / 21	FLOAT	MAX_TRAVEL_RANGE_POS The maximum travel in positive direction (user unit)	"Soft limit", based on the home (zero) position. If the soft limit is smaller than the position value for the positive limit switch (which is given by the sum of the parameters 0x16 and 0x2F), the positive limit switch can not be used for referencing. Can be negative.

Para-meter ID (hexa-decimal / decimal)	Data Type	Parameter Description	Possible Values/Notes
0x16 / 22	FLOAT	VALUE_AT_REF_POS The position value at the reference position (user unit)	The position value which is to be set when the mechanics performs a reference move to the reference switch. Must be set even if no reference switch is present in the mechanics because it is used to to calculate the position values to be set after reference moves to the limit switches.
0x17 / 23	FLOAT	DISTANCE_REF_TO_N_LIM The distance between reference switch and negative limit switch (user unit)	Represents the physical distance between the reference switch and the negative limit switch integrated in the mechanics. Must be set even if no reference switch is present in the mechanics because the position is set to the difference of VALUE_AT_REF_POS and DISTANCE_REF_TO_N_LIM when the mechanics performs a reference move to the negative limit switch.
0x18 / 24	FLOAT	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
0x19 / 25	FLOAT	Stage type	0 = DC motor 2 = Voice coil
0x1A / 26	FLOAT	Stage has brakes	0 = Stage has no brakes 1 = Stage has brakes
0x2F / 47	FLOAT	DISTANCE_REF_TO_P_LIM The distance between reference switch and positive limit switch (user unit)	Represents the physical distance between the reference switch and the positive limit switch integrated in the mechanics. Must be set even if no reference switch is present in the mechanics because the position is set to the sum of VALUE_AT_REF_POS and DISTANCE_REF_TO_P_LIM when the mechanics performs a reference move to the positive limit switch.
0x30 / 48	FLOAT	MAX_TRAVEL_RANGE_NEG The maximum travel in negative direction (user unit)	"Soft limit", based on the home (zero) position. If the soft limit is larger than the position value for the negative limit switch (which is given by the difference of the parameters 0x16 and 0x17), the negative limit switch can not be used for referencing. Can be negative.
0x31 / 49	FLOAT	Invert the reference	1 = invert the reference, else 0
0x32 / 50	FLOAT	Stage has limit switches; enables / disables the stopping of the motion at the limit switches	0 = Stage has limit switches 1 = Stage has no limit switches

Para-meter ID (hexa-decimal / decimal)	Data Type	Parameter Description	Possible Values/Notes
0x36 / 54	FLOAT	Settle window (counts)	0 to 2^{31} The settle window is centered around the target position. The on-target status becomes "true" when the current position stays in this window for at least the settle time (parameter 0x3F).
0x3C / 60	FLOAT	Stage name	Maximum 31 characters
0x3F / 63	FLOAT	Settle time (s)	0.000 to 1.000 s; Used for on-target detection: The on-target status becomes "true" when the current position stays in the settle window (parameter 0x36) for at least the settle time. If the settle time is set to 0, then the axis is on target when the trajectory has finished, irrespective of the current position.
0x40 / 64	FLOAT	Hold current (mA)	When motion has finished, after a given delay time (parameter 0x42) the hold current is applied. Note that normally the hold current is about 25% of the drive current which allows to keep the temperature of the stepper motor down, close to room temperature.
0x41 / 65	FLOAT	Drive current (mA)	Gives the motor phase current (drive current) for moving state.
0x42 / 66	FLOAT	Hold time (ms)	Gives the hold time, that is the delay time between completion of a move and the activation of the hold current.
0x43 / 67	FLOAT	Max current (mA)	Maximum value for hold current (parameter 0x40) and drive current (parameter 0x41)
0x49 / 73	FLOAT	Current closed-loop velocity (user unit/s)	Gives the current velocity, limited by parameter 0xA Can also be changed by Mercury_VEL()
0x4A / 74	FLOAT	Maximum closed-loop acceleration (user unit/s^2)	Gives the maximum value for parameter 0xB
0x50 / 80	FLOAT	Velocity for reference move (user unit/s)	Gives the maximum velocity to be used for reference moves with Mercury_REF(), Mercury_MPL(), Mercury_MNL(); if set to 0, reference moves are not possible
0x94 / 148	FLOAT	Notch filter frequency (Hz)	40 to 10000 The corresponding frequency component in the control value is reduced to compensate for unwanted resonances in the mechanics. Only active in closed-loop operation. Should normally not be changed (try to change only with very high loads).
0x95 / 149	FLOAT	Notch filter edge	0.4 to 10 Gives the slope of the filter edge. Do not change.

Para-meter ID (hexa-decimal / decimal)	Data Type	Parameter Description	Possible Values/Notes
0x100 / 256	FLOAT	Trackball mode	0 = trackball disabled 1 = trackball enabled Connect the digital TTL signals A and B (also referred to as quadrature signals) provided by the trackball to the digital input lines 3 and 4 of the "I/O" socket (pinout see User manual) on the Mercury™ controller. The lines are terminated by 10k to GND. While the trackball mode is active, move commands or joystick control are not accepted.
0x101 / 257	FLOAT	Trackball increment (counts)	Gives the increment length for trackball operation, minimum value = 1. Each signal transition on the trackball lines will shift the target by the given number of counts if the trackball is enabled by parameter 0x100.
0x102 / 258	FLOAT	Trackball filter	0 to 255 The parameter determines how often the same signal level must be read before the signal transition on the trackball lines is accepted. This suppresses transient noise and allows for stable reading. If 0, the filter is disabled.
0x110 / 272	FLOAT	Control mode	Gives the control mode to be applied, see C-863 User manual for details 0 = Position control (force control is OFF) 1 = Force control (position control is OFF) 2 = Position control and force control are ONs
0x111 / 273	FLOAT	Force control target	0 to 1024 Used in “force control” or “position and force control” mode where it defines the force target that shall be reached and maintained by varying the position.
0x112 / 274	FLOAT	P-term for force control algorithm	0 to 65,535
0x113 / 275	FLOAT	I-term for force control algorithm	0 to 65,535
0x114 / 276	FLOAT	I-limit for force control algorithm	0 to 65,535
0x115 / 277	FLOAT	D-term for force control algorithm	0 to 65,535
0x116 / 278	FLOAT	Offset to force sensor input	Gives an offset to the input of the additional (force) sensor. Note that this offset is already included if you query the current value of the additional sensor via parameter 0x11A.
0x117 / 279	FLOAT	Low pass filter for force sensor input (Hz)	40 to 10000 Gives the frequency of the low pass filter on the input of the additional (force) sensor used for the “force control” or “position and force control” modes.

Para-meter ID (hexa-decimal / decimal)	Data Type	Parameter Description	Possible Values/Notes
0x11A / 282	FLOAT	Current force read-only parameter	-2048 to 2048 Current value of the additional sensor used for the force control loop, i.e. the input signal on pin 2 of the "Joystick" socket. The input voltage range is -10 to +10 V. The voltage value is converted by a 12 bit A/D converter. The value reported includes the offset set with parameter 0x116, and was already filtered by the low pass filter set with parameter 0x117. Furthermore, any sign settings made with the FS (Set Force Sign) command of the native command set are already applied to the signal (see MercuryNativeCommands Manual MS176E for details).
0x120 / 288	FLOAT	Joystick offset	Gives an offset to the analog input provided by the joystick. This allows to correct the neutral (center) value of the joystick reading, e.g. if the actual joystick reading is 120, with bias 8 the reading used would be 128.

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	0xE	Numerator of the counts per physical unit factor
CpuD	0xF	Denominator of the counts per physical unit factor
SF	0x12	Scaling factor**

*Number means the parameter ID in Mercury_SPA (p. 45) and Mercury_qSPA (p. 40) and in the list in Section 55.

**See also Mercury_DFF (p. 27).

The "Counts per physical unit factor" which results from parameter 0xE divided by parameter 0xF includes the physical transmission ratio and the resolution of the stage.

CAUTION

To customize the physical unit of a stage do not change parameter 0xE and parameter 0xF but use Mercury_DFF (p. 27) instead. Although Mercury_DFF has the same effect as changing parameter 0x12 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.

11.4. Travel Range Adjustment

The figures below give a universal hardware scheme of a positioning stage with incremental sensor, reference and limit switches. To work with such a stage, the parameters of the PI Mercury GCS DLL must be adjusted properly (see "Parameter Handling" on p. 55 for how to modify parameter values).

In the example shown in the first figure, the travel range, i.e. the distance from negative to positive limit switch is 20 mm, the distance between the negative limit switch and the reference switch is 8 mm, and the distance between reference switch and positive limit switch is 12 mm. These hardware properties are represented by the following parameters in the PI Mercury GCS DLL:

DISTANCE_REF_TO_N_LIM (parameter ID 0x17) = 8

DISTANCE_REF_TO_P_LIM (parameter ID 0x2F) = 12

To allow for flexible localization of the home position (0), a special parameter is provided. It gives the offset between reference switch and home position which is to be valid for the stage after a reference move (see below). In the example, the home position is to be located at the negative limit switch after a reference move, and hence the offset between reference switch and home position is 8 mm.

VALUE_AT_REF_POS (parameter ID 0x16) = 8

To allow for absolute moves, either an absolute "initial" position can be set with Mercury_POS(), or the stage can perform a reference move to a known position where a defined position value will be set as the current position (see "Referencing" on p. 11 for further details). By default, a reference move is required. In the example, known positions for reference moves are given by the reference switch and the limit switches. Depending on the switch used for the reference move, a certain combination of the above-mentioned parameters is used to calculate the position to be set at the end of the move:

- Reference switch (Mercury_REF()): the stage is moved to the reference switch, and the value of VALUE_AT_REF_POS is set as the current position.
- Negative limit switch (Mercury_MNL()): the stage is moved to the negative limit switch and the difference of VALUE_AT_REF_POS and DISTANCE_REF_TO_N_LIM is set as the current position (can be negative).
- Positive limit switch (Mercury_MPL()): the stage is moved to the positive limit switch and the sum of VALUE_AT_REF_POS and DISTANCE_REF_TO_P_LIM is set as the current position.

It is furthermore possible to set "soft limits" which establish a "safety distance" which the stage will not enter on both ends of the travel range. Those soft limits always refer to the current home position (0; in the example located at the negative limit switch after a reference move). The soft limits are to be deactivated in the example so that the corresponding parameters must be as follows:

MAX_TRAVEL_RANGE_POS (parameter ID 0x15) = 20 mm

MAX_TRAVEL_RANGE_NEG (parameter ID 0x30) = 0 mm

(This means that the stage can move 20 mm in positive direction, starting from the home position, and 0 mm in negative direction, starting from the home position.)

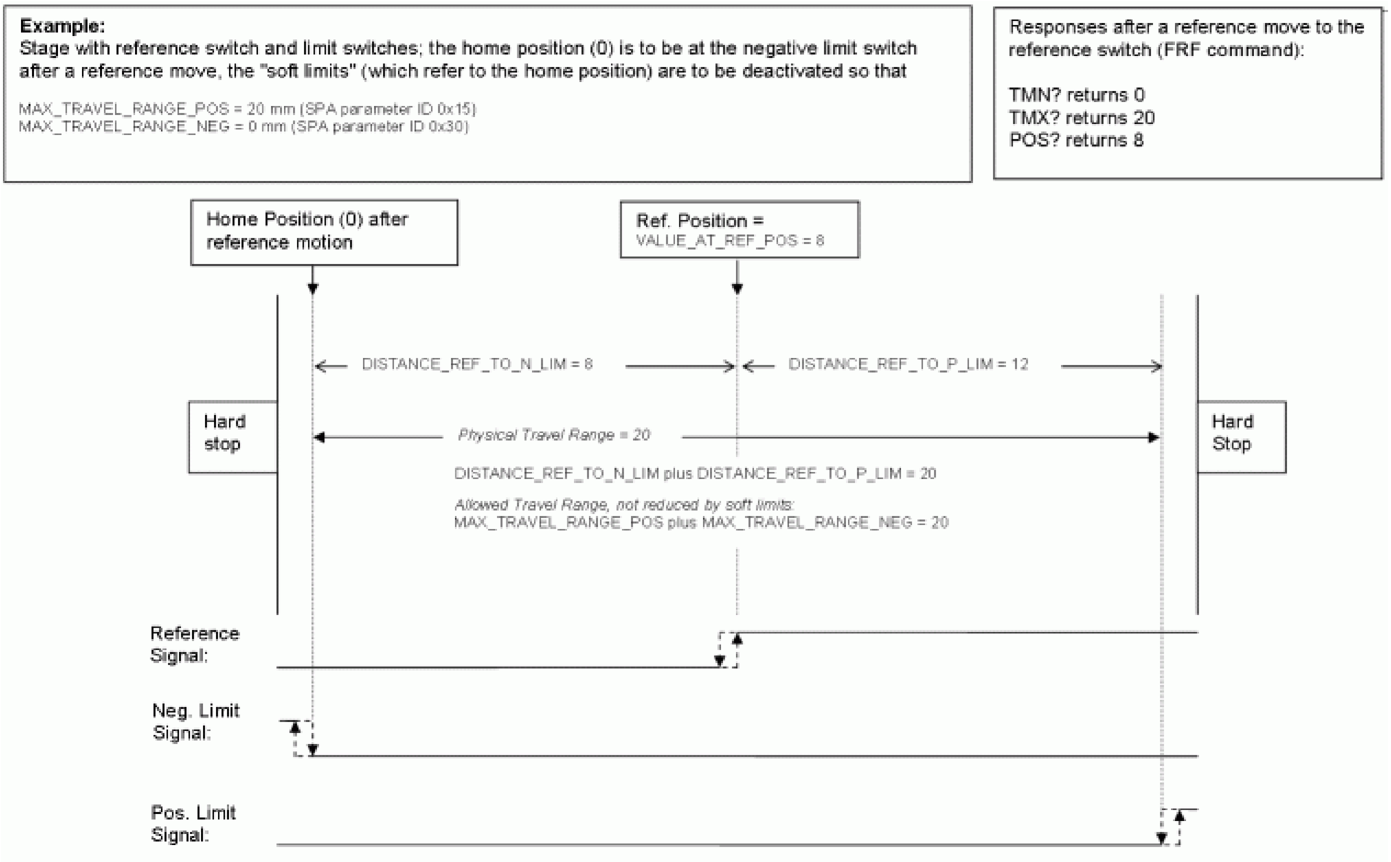


Figure 1: Positioning stage and corresponding controller parameters

Now in the same example, a "safety distance" is to be established on both ends of the travel range by setting soft limits, and the home position is to be located at about 1/3 of the distance between the new negative end of the travel range and the reference switch. The limit switches can not be used for reference moves anymore.

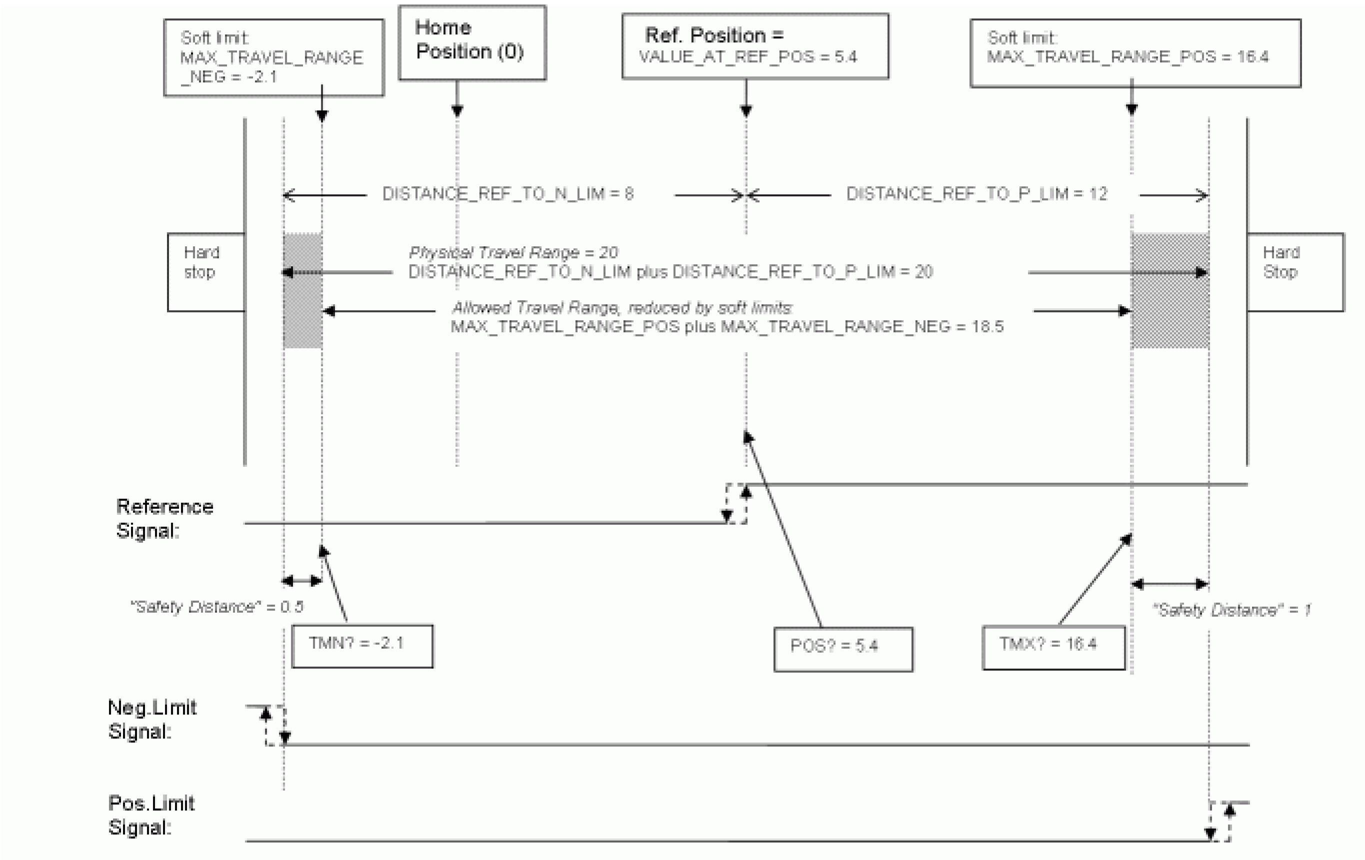


Figure 2: Positioning stage, soft limits set in the controller to reduce the travel range

After the stage was referenced again by moving it to the reference switch (Mercury_REF()), the following responses will be given:

Mercury_qTMN() returns -2.1
Mercury_qTMX() returns 16.4
Mercury_qPOS() returns 5.4

CAUTION

If the soft limits (MAX_TRAVEL_RANGE_POS and MAX_TRAVEL_RANGE_NEG) are used to reduce the travel range, the limit switches can not be used for reference moves. Mercury_MNL and Mercury_MPL will provoke an error message, and only the reference switch can be used for a reference move (Mercury_REF).

Be careful when setting the values for VALUE_AT_REF_POS, MAX_TRAVEL_RANGE_POS and MAX_TRAVEL_RANGE_NEG because there is no plausibility check.

The soft limits may not be outside of the physical travel range: MAX_TRAVEL_RANGE_POS ≤ DISTANCE_REF_TO_P_LIM + VALUE_AT_REF_POS MAX_TRAVEL_RANGE_NEG ≥ VALUE_AT_REF_POS - DISTANCE_REF_TO_N_LIM Otherwise, reference moves to the limit switches would have incorrect results because the values of the soft limits would be set at the end of the referencing procedure.

Be careful when referencing the stage by setting an initial absolute position with Mercury_POS() since the values for MAX_TRAVEL_RANGE_POS and MAX_TRAVEL_RANGE_NEG are not adapted. In the worst case, the soft limits will now be outside of the physical travel range, and the stage will no longer be able to move since the move commands check the soft limit settings.

12. Error Codes

The error codes listed here are those of the *PI General Command Set.* As such, some are not relevant to Mercury™ controllers and will simply never occur with the systems this manual describes.

Controller Errors

0	PI_CNTR_NO_ERROR	No error
1	PI_CNTR_PARAM_SYNTAX	Parameter syntax error
2	PI_CNTR_UNKNOWN_COMMAND	Unknown command
3	PI_CNTR_COMMAND_TOO_LONG	Command length out of limits or command buffer overrun
4	PI_CNTR_SCAN_ERROR	Error while scanning
5	PI_CNTR_MOVE_WITHOUT_REF_OR_NO_SERVO	Unallowable move attempted on unreferenced axis, or move attempted with servo off
6	PI_CNTR_INVALID_SGA_PARAM	Parameter for SGA not valid
7	PI_CNTR_POS_OUT_OF_LIMITS	Position out of limits
8	PI_CNTR_VEL_OUT_OF_LIMITS	Velocity out of limits
9	PI_CNTR_SET_PIVOT_NOT_POSSIBLE	Attempt to set pivot point while U,V and W not all 0
10	PI_CNTR_STOP	Controller was stopped by command
11	PI_CNTR_SST_OR_SCAN_RANGE	Parameter for SST or for one of the embedded scan algorithms out of range
12	PI_CNTR_INVALID_SCAN_AXES	Invalid axis combination for fast scan
13	PI_CNTR_INVALID_NAV_PARAM	Parameter for NAV out of range
14	PI_CNTR_INVALID_ANALOG_INPUT	Invalid analog channel
15	PI_CNTR_INVALID_AXIS_IDENTIFIER	Invalid axis identifier
16	PI_CNTR_INVALID_STAGE_NAME	Unknown stage name
17	PI_CNTR_PARAM_OUT_OF_RANGE	Parameter out of range
18	PI_CNTR_INVALID_MACRO_NAME	Invalid macro name
19	PI_CNTR_MACRO_RECORD	Error while recording macro
20	PI_CNTR_MACRO_NOT_FOUND	Macro not found

21	PI_CNTR_AXIS_HAS_NO_BRAKE	Axis has no brake
22	PI_CNTR_DOUBLE_AXIS	Axis identifier specified more than once
23	PI_CNTR_ILLEGAL_AXIS	Illegal axis
24	PI_CNTR_PARAM_NR	Incorrect number of parameters
25	PI_CNTR_INVALID_REAL_NR	Invalid floating point number
26	PI_CNTR_MISSING_PARAM	Parameter missing
27	PI_CNTR_SOFT_LIMIT_OUT_OF_RANGE	Soft limit out of range
28	PI_CNTR_NO_MANUAL_PAD	No manual pad found
29	PI_CNTR_NO_JUMP	No more step-response values
30	PI_CNTR_INVALID_JUMP	No step-response values recorded
31	PI_CNTR_AXIS_HAS_NO_REFERENCE	Axis has no reference sensor
32	PI_CNTR_STAGE_HAS_NO_LIM_SWITCH	Axis has no limit switch
33	PI_CNTR_NO_RELAY_CARD	No relay card installed
34	PI_CNTR_CMD_NOT_ALLOWED_FOR_STAGE	Command not allowed for selected stage(s)
35	PI_CNTR_NO_DIGITAL_INPUT	No digital input installed
36	PI_CNTR_NO_DIGITAL_OUTPUT	No digital output configured
37	PI_CNTR_NO_MCM	No more MCM responses
38	PI_CNTR_INVALID_MCM	No MCM values recorded
39	PI_CNTR_INVALID_CNTR_NUMBER	Controller number invalid
40	PI_CNTR_NO_JOYSTICK_CONNECTED	No joystick configured
41	PI_CNTR_INVALID_EGE_AXIS	Invalid axis for electronic gearing, axis can not be slave
42	PI_CNTR_SLAVE_POSITION_OUT_OF_RANGE	Position of slave axis is out of range

43	PI_CNTR_COMMAND_EGE_SLAVE	Slave axis cannot be commanded directly when electronic gearing is enabled
44	PI_CNTR_JOYSTICK_CALIBRATION_FAILED	Calibration of joystick failed
45	PI_CNTR_REFERENCING_FAILED	Referencing failed
46	PI_CNTR_OPM_MISSING	OPM (Optical Power Meter) missing
47	PI_CNTR_OPM_NOT_INITIALIZED	OPM (Optical Power Meter) not initialized or cannot be initialized
48	PI_CNTR_OPM_COM_ERROR	OPM (Optical Power Meter) Communication Error
49	PI_CNTR_MOVE_TO_LIMIT_SWITCH_FAILED	Move to limit switch failed
50	PI_CNTR_REF_WITH_REF_DISABLED	Attempt to reference axis with referencing disabled
51	PI_CNTR_AXIS_UNDER_JOYSTICK_CONTROL	Selected axis is controlled by joystick
52	PI_CNTR_COMMUNICATION_ERROR	Controller detected communication error
53	PI_CNTR_DYNAMIC_MOVE_IN_PROCESS	MOV! motion still in progress
54	PI_CNTR_UNKNOWN_PARAMETER	Unknown parameter
55	PI_CNTR_NO_REP_RECORDED	No commands were recorded with REP
56	PI_CNTR_INVALID_PASSWORD	Password invalid
57	PI_CNTR_INVALID_RECORDER_CHAN	Data Record Table does not exist
58	PI_CNTR_INVALID_RECORDER_SRC_OPT	Source does not exist; number too low or too high
59	PI_CNTR_INVALID_RECORDER_SRC_CHAN	Source Record Table number too low or too high
60	PI_CNTR_PARAM_PROTECTION	Protected Param: current Command Level (CCL) too low
61	PI_CNTR_AUTOZERO_RUNNING	Command execution not possible while Autozero is running
62	PI_CNTR_NO_LINEAR_AXIS	Autozero requires at least one linear axis
63	PI_CNTR_INIT_RUNNING	Initialization still in progress

64	PI_CNTR_READ_ONLY_PARAMETER	Parameter is read-only
65	PI_CNTR_PAM_NOT_FOUND	Parameter not found in non-volatile memory
66	PI_CNTR_VOL_OUT_OF_LIMITS	Voltage out of limits
67	PI_CNTR_WAVE_TOO_LARGE	Not enough memory available for requested wave curve
68	PI_CNTR_NOT_ENOUGH_DDL_MEMORY	Not enough memory available for DDL table; DDL can not be started
69	PI_CNTR_DDL_TIME_DELAY_TOO_LARGE	Time delay larger than DDL table; DDL can not be started
70	PI_CNTR_DIFFERENT_ARRAY_LENGTH	The requested arrays have different lengths; query them separately
71	PI_CNTR_GEN_SINGLE_MODE_RESTART	Attempt to restart the generator while it is running in single step mode
72	PI_CNTR_ANALOG_TARGET_ACTIVE	Motion commands and wave generator activation are not allowed when analog target is active
73	PI_CNTR_WAVE_GENERATOR_ACTIVE	Motion commands are not allowed when wave generator is active
74	PI_CNTR_AUTOZERO_DISABLED	No sensor channel or no piezo channel connected to selected axis (sensor and piezo matrix)
75	PI_CNTR_NO_WAVE_SELECTED	Generator started (WGO) without having selected a wave table (WSL).
76	PI_CNTR_IF_BUFFER_OVERRUN	Interface buffer did overrun and command couldn't be received correctly
77	PI_CNTR_NOT_ENOUGH_RECORDED_DATA	Data Record Table does not hold enough recorded data
78	PI_CNTR_TABLE_DEACTIVATED	Data Record Table is not configured for recording
79	PI_CNTR_OPENLOOP_VALUE_SET_WHEN_SERVO_ON	Open-loop commands (SVA, SVR) are not allowed when servo is on
80	PI_CNTR_RAM_ERROR	Hardware error affecting RAM
81	PI_CNTR_MACRO_UNKNOWN_COMMAND	Not macro command

82	PI_CNTR_MACRO_PC_ERROR	Macro counter out of range
83	PI_CNTR_JOYSTICK_ACTIVE	Joystick is active
84	PI_CNTR_MOTOR_IS_OFF	Motor is off
85	PI_CNTR_ONLY_IN_MACRO	Macro-only command
86	PI_CNTR_JOYSTICK_UNKNOWN_AXIS	Invalid joystick axis
87	PI_CNTR_JOYSTICK_UNKNOWN_ID	Joystick unknown
88	PI_CNTR_REF_MODE_IS_ON	Move without referenced stage
89	PI_CNTR_NOT_ALLOWED_IN_CURRENT_MOTION_MODE	Command not allowed in current motion mode
90	PI_CNTR_DIO_AND_TRACING_NOT_POSSIBLE	No tracing possible while digital IOs are used on this HW revision. Reconnect to switch operation mode.
91	PI_CNTR_COLLISION	Move not possible, would cause collision
100	PI_LABVIEW_ERROR	PI LabVIEW driver reports error. See source control for details.
200	PI_CNTR_NO_AXIS	No stage connected to axis
201	PI_CNTR_NO_AXIS_PARAM_FILE	File with axis parameters not found
202	PI_CNTR_INVALID_AXIS_PARAM_FILE	Invalid axis parameter file
203	PI_CNTR_NO_AXIS_PARAM_BACKUP	Backup file with axis parameters not found
204	PI_CNTR_RESERVED_204	PI internal error code 204
205	PI_CNTR_SMO_WITH_SERVO_ON	SMO with servo on
206	PI_CNTR_UUDECODE_INCOMPLETE_HEADER	uudecode: incomplete header
207	PI_CNTR_UUDECODE_NOTHING_TO_DECODE	uudecode: nothing to decode
208	PI_CNTR_UUDECODE_ILLEGAL_FORMAT	uudecode: illegal UUE format
209	PI_CNTR_CRC32_ERROR	CRC32 error
210	PI_CNTR_ILLEGAL_FILENAME	Illegal file name (must be 8-0 format)

211	PI_CNTR_FILE_NOT_FOUND	File not found on controller
212	PI_CNTR_FILE_WRITE_ERROR	Error writing file on controller
213	PI_CNTR_DTR_HINDERS_VELOCITY_CHANGE	VEL command not allowed in DTR Command Mode
214	PI_CNTR_POSITION_UNKNOWN	Position calculations failed
215	PI_CNTR_CONN_POSSIBLY_BROKEN	The connection between controller and stage may be broken
216	PI_CNTR_ON_LIMIT_SWITCH	The connected stage has driven into a limit switch, some controllers need CLR to resume operation
217	PI_CNTR_UNEXPECTED_STRUT_STOP	Strut test command failed because of an unexpected strut stop
218	PI_CNTR_POSITION_BASED_ON_ESTIMATION	While MOV! is running position can only be estimated!
219	PI_CNTR_POSITION_BASED_ON_INTERPOLATION	Position was calculated during MOV motion
230	PI_CNTR_INVALID_HANDLE	Invalid handle
231	PI_CNTR_NO_BIOS_FOUND	No bios found
232	PI_CNTR_SAVE_SYS_CFG_FAILED	Save system configuration failed
233	PI_CNTR_LOAD_SYS_CFG_FAILED	Load system configuration failed
301	PI_CNTR_SEND_BUFFER_OVERFLOW	Send buffer overflow
302	PI_CNTR_VOLTAGE_OUT_OF_LIMITS	Voltage out of limits
303	PI_CNTR_OPEN_LOOP_MOTION_SET_WHEN_SERVO_ON	Open-loop motion attempted when servo ON
304	PI_CNTR_RECEIVING_BUFFER_OVERFLOW	Received command is too long
305	PI_CNTR_EEPROM_ERROR	Error while reading/writing EEPROM
306	PI_CNTR_I2C_ERROR	Error on I2C bus
307	PI_CNTR_RECEIVING_TIMEOUT	Timeout while receiving command
308	PI_CNTR_TIMEOUT	A lengthy operation has not finished in the expected time

309	PI_CNTR_MACRO_OUT_OF_SPACE	Insufficient space to store macro
310	PI_CNTR_EUI_OLDVERSION_CFGDATA	Configuration data has old version number
311	PI_CNTR_EUI_INVALID_CFGDATA	Invalid configuration data
333	PI_CNTR_HARDWARE_ERROR	Internal hardware error
400	PI_CNTR_WAV_INDEX_ERROR	Wave generator index error
401	PI_CNTR_WAV_NOT_DEFINED	Wave table not defined
402	PI_CNTR_WAV_TYPE_NOT_SUPPORTED	Wave type not supported
403	PI_CNTR_WAV_LENGTH_EXCEEDS_LIMIT	Wave length exceeds limit
404	PI_CNTR_WAV_PARAMETER_NR	Wave parameter number error
405	PI_CNTR_WAV_PARAMETER_OUT_OF_LIMIT	Wave parameter out of range
406	PI_CNTR_WGO_BIT_NOT_SUPPORTED	WGO command bit not supported
502	PI_CNTR_REDUNDANCY_LIMIT_EXCEEDED	Position consistency check failed
503	PI_CNTR_COLLISION_SWITCH_ACTIVATED	Hardware collision sensor(s) are activated
504	PI_CNTR_FOLLOWING_ERROR	Strut following error occurred, e.g. caused by overload or encoder failure
555	PI_CNTR_UNKNOWN_ERROR	BasMac: unknown controller error
601	PI_CNTR_NOT_ENOUGH_MEMORY	not enough memory
602	PI_CNTR_HW_VOLTAGE_ERROR	hardware voltage error
603	PI_CNTR_HW_TEMPERATURE_ERROR	hardware temperature out of range
1000	PI_CNTR_TOO_MANY_NESTED_MACROS	Too many nested macros
1001	PI_CNTR_MACRO_ALREADY_DEFINED	Macro already defined
1002	PI_CNTR_NO_MACRO_RECORDING	Macro recording not activated
1003	PI_CNTR_INVALID_MAC_PARAM	Invalid parameter for MAC

1004	PI_CNTR_RESERVED_1004	PI internal error code 1004
1005	PI_CNTR_CONTROLLER_BUSY	Controller is busy with some lengthy operation (e.g. reference move, fast scan algorithm)
2000	PI_CNTR_ALREADY_HAS_SERIAL_NUMBER	Controller already has a serial number
4000	PI_CNTR_SECTOR_ERASE_FAILED	Sector erase failed
4001	PI_CNTR_FLASH_PROGRAM_FAILED	Flash program failed
4002	PI_CNTR_FLASH_READ_FAILED	Flash read failed
4003	PI_CNTR_HW_MATCHCODE_ERROR	HW match code missing/invalid
4004	PI_CNTR_FW_MATCHCODE_ERROR	FW match code missing/invalid
4005	PI_CNTR_HW_VERSION_ERROR	HW version missing/invalid
4006	PI_CNTR_FW_VERSION_ERROR	FW version missing/invalid
4007	PI_CNTR_FW_UPDATE_ERROR	FW update failed
5200	PI_CNTR_AXIS_NOT_CONFIGURED	Axis must be configured for this action

Interface Errors

0	COM_NO_ERROR	No error occurred during function call
-1	COM_ERROR	Error during com operation (could not be specified)
-2	SEND_ERROR	Error while sending data
-3	REC_ERROR	Error while receiving data
-4	NOT_CONNECTED_ERROR	Not connected (no port with given ID open)
-5	COM_BUFFER_OVERFLOW	Buffer overflow
-6	CONNECTION_FAILED	Error while opening port
-7	COM_TIMEOUT	Timeout error

-8	COM_MULTILINE_RESPONSE	There are more lines waiting in buffer
-9	COM_INVALID_ID	There is no interface or DLL handle with the given ID
-10	COM_NOTIFY_EVENT_ERROR	Event/message for notification could not be opened
-11	COM_NOT_IMPLEMENTED	Function not supported by this interface type
-12	COM_ECHO_ERROR	Error while sending "echoed" data
-13	COM_GPIB_EDVR	IEEE488: System error
-14	COM_GPIB_ECIC	IEEE488: Function requires GPIB board to be CIC
-15	COM_GPIB_ENOL	IEEE488: Write function detected no listeners
-16	COM_GPIB_EADR	IEEE488: Interface board not addressed correctly
-17	COM_GPIB_EARG	IEEE488: Invalid argument to function call
-18	COM_GPIB_ESAC	IEEE488: Function requires GPIB board to be SAC
-19	COM_GPIB_EABO	IEEE488: I/O operation aborted
-20	COM_GPIB_ENEB	IEEE488: Interface board not found
-21	COM_GPIB_EDMA	IEEE488: Error performing DMA
-22	COM_GPIB_EOIP	IEEE488: I/O operation started before previous operation completed
-23	COM_GPIB_ECAP	IEEE488: No capability for intended operation
-24	COM_GPIB_EFSO	IEEE488: File system operation error
-25	COM_GPIB_EBUS	IEEE488: Command error during device call
-26	COM_GPIB_ESTB	IEEE488: Serial poll-status byte lost
-27	COM_GPIB_ESRQ	IEEE488: SRQ remains asserted
-28	COM_GPIB_ETAB	IEEE488: Return buffer full

-29	COM_GPIB_ELCK	IEEE488: Address or board locked
-30	COM_RS_INVALID_DATA_BITS	RS-232: 5 data bits with 2 stop bits is an invalid combination, as is 6, 7, or 8 data bits with 1.5 stop bits
-31	COM_ERROR_RS_SETTINGS	RS-232: Error configuring the COM port
-32	COM_INTERNAL_RESOURCES_ERROR	Error dealing with internal system resources (events, threads, ...)
-33	COM_DLL_FUNC_ERROR	A DLL or one of the required functions could not be loaded
-34	COM_FTDIUSB_INVALID_HANDLE	FTDIUSB: invalid handle
-35	COM_FTDIUSB_DEVICE_NOT_FOUND	FTDIUSB: device not found
-36	COM_FTDIUSB_DEVICE_NOT_OPENED	FTDIUSB: device not opened
-37	COM_FTDIUSB_IO_ERROR	FTDIUSB: IO error
-38	COM_FTDIUSB_INSUFFICIENT_RESOURCES	FTDIUSB: insufficient resources
-39	COM_FTDIUSB_INVALID_PARAMETER	FTDIUSB: invalid parameter
-40	COM_FTDIUSB_INVALID_BAUD_RATE	FTDIUSB: invalid baud rate
-41	COM_FTDIUSB_DEVICE_NOT_OPENED_FOR_ERASE	FTDIUSB: device not opened for erase
-42	COM_FTDIUSB_DEVICE_NOT_OPENED_FOR_WRITE	FTDIUSB: device not opened for write
-43	COM_FTDIUSB_FAILED_TO_WRITE_DEVICE	FTDIUSB: failed to write device
-44	COM_FTDIUSB_EEPROM_READ_FAILED	FTDIUSB: EEPROM read failed
-45	COM_FTDIUSB_EEPROM_WRITE_FAILED	FTDIUSB: EEPROM write failed
-46	COM_FTDIUSB_EEPROM_ERASE_FAILED	FTDIUSB: EEPROM erase failed
-47	COM_FTDIUSB_EEPROM_NOT_PRESENT	FTDIUSB: EEPROM not present
-48	COM_FTDIUSB_EEPROM_NOT_PROGRAMMED	FTDIUSB: EEPROM not programmed
-49	COM_FTDIUSB_INVALID_ARGS	FTDIUSB: invalid arguments

-50	COM_FTDIUSB_NOT_SUPPORTED	FTDIUSB: not supported
-51	COM_FTDIUSB_OTHER_ERROR	FTDIUSB: other error
-52	COM_PORT_ALREADY_OPEN	Error while opening the COM port: was already open
-53	COM_PORT_CHECKSUM_ERROR	Checksum error in received data from COM port
-54	COM_SOCKET_NOT_READY	Socket not ready, you should call the function again
-55	COM_SOCKET_PORT_IN_USE	Port is used by another socket
-56	COM_SOCKET_NOT_CONNECTED	Socket not connected (or not valid)
-57	COM_SOCKET_TERMINATED	Connection terminated (by peer)
-58	COM_SOCKET_NO_RESPONSE	Can't connect to peer
-59	COM_SOCKET_INTERRUPTED	Operation was interrupted by a nonblocked signal
-60	COM_PCI_INVALID_ID	No device with this ID is present
-61	COM_PCI_ACCESS_DENIED	Driver could not be opened (on Vista: run as administrator!)

DLL Errors

-1001	PI_UNKNOWN_AXIS_IDENTIFIER	Unknown axis identifier
-1002	PI_NR_NAV_OUT_OF_RANGE	Number for NAV out of range--must be in [1,10000]
-1003	PI_INVALID_SGA	Invalid value for SGA--must be one of 1, 10, 100, 1000
-1004	PI_UNEXPECTED_RESPONSE	Controller sent unexpected response
-1005	PI_NO_MANUAL_PAD	No manual control pad installed, calls to SMA and related commands are not allowed
-1006	PI_INVALID_MANUAL_PAD_KNOB	Invalid number for manual control pad knob
-1007	PI_INVALID_MANUAL_PAD_AXIS	Axis not currently controlled by a manual control pad

-1008	PI_CONTROLLER_BUSY	Controller is busy with some lengthy operation (e.g. reference move, fast scan algorithm)
-1009	PI_THREAD_ERROR	Internal error--could not start thread
-1010	PI_IN_MACRO_MODE	Controller is (already) in macro mode--command not valid in macro mode
-1011	PI_NOT_IN_MACRO_MODE	Controller not in macro mode--command not valid unless macro mode active
-1012	PI_MACRO_FILE_ERROR	Could not open file to write or read macro
-1013	PI_NO_MACRO_OR_EMPTY	No macro with given name on controller, or macro is empty
-1014	PI_MACRO_EDITOR_ERROR	Internal error in macro editor
-1015	PI_INVALID_ARGUMENT	One or more arguments given to function is invalid (empty string, index out of range, ...)
-1016	PI_AXIS_ALREADY_EXISTS	Axis identifier is already in use by a connected stage
-1017	PI_INVALID_AXIS_IDENTIFIER	Invalid axis identifier
-1018	PI_COM_ARRAY_ERROR	Could not access array data in COM server
-1019	PI_COM_ARRAY_RANGE_ERROR	Range of array does not fit the number of parameters
-1020	PI_INVALID_SPA_CMD_ID	Invalid parameter ID given to SPA or SPA?
-1021	PI_NR_AVG_OUT_OF_RANGE	Number for AVG out of range--must be >0
-1022	PI_WAV_SAMPLES_OUT_OF_RANGE	Incorrect number of samples given to WAV
-1023	PI_WAV_FAILED	Generation of wave failed
-1024	PI_MOTION_ERROR	Motion error while axis in motion, call CLR to resume operation
-1025	PI_RUNNING_MACRO	Controller is (already) running a macro
-1026	PI_PZT_CONFIG_FAILED	Configuration of PZT stage or amplifier failed
-1027	PI_PZT_CONFIG_INVALID_PARAMS	Current settings are not valid for desired configuration

-1028	PI_UNKNOWN_CHANNEL_IDENTIFIER	Unknown channel identifier
-1029	PI_WAVE_PARAM_FILE_ERROR	Error while reading/writing wave generator parameter file
-1030	PI_UNKNOWN_WAVE_SET	Could not find description of wave form. Maybe WG.INI is missing?
-1031	PI_WAVE_EDITOR_FUNC_NOT_LOADED	The WGWaveEditor DLL function was not found at startup
-1032	PI_USER_CANCELLED	The user cancelled a dialog
-1033	PI_C844_ERROR	Error from C-844 Controller
-1034	PI_DLL_NOT_LOADED	DLL necessary to call function not loaded, or function not found in DLL
-1035	PI_PARAMETER_FILE_PROTECTED	The open parameter file is protected and cannot be edited
-1036	PI_NO_PARAMETER_FILE_OPENED	There is no parameter file open
-1037	PI_STAGE_DOES_NOT_EXIST	Selected stage does not exist
-1038	PI_PARAMETER_FILE_ALREADY_OPENED	There is already a parameter file open. Close it before opening a new file
-1039	PI_PARAMETER_FILE_OPEN_ERROR	Could not open parameter file
-1040	PI_INVALID_CONTROLLER_VERSION	The version of the connected controller is invalid
-1041	PI_PARAM_SET_ERROR	Parameter could not be set with SPA--parameter not defined for this controller!
-1042	PI_NUMBER_OF_POSSIBLE_WAVES_EXCEEDED	The maximum number of wave definitions has been exceeded
-1043	PI_NUMBER_OF_POSSIBLE_GENERATORS_EXCEEDED	The maximum number of wave generators has been exceeded
-1044	PI_NO_WAVE_FOR_AXIS_DEFINED	No wave defined for specified axis
-1045	PI_CANT_STOP_OR_START_WAV	Wave output to axis already stopped/started
-1046	PI_REFERENCE_ERROR	Not all axes could be referenced
-1047	PI_REQUIRED_WAVE_NOT_FOUND	Could not find parameter set required by frequency relation

-1048	PI_INVALID_SPP_CMD_ID	Command ID given to SPP or SPP? is not valid
-1049	PI_STAGE_NAME_ISNT_UNIQUE	A stage name given to CST is not unique
-1050	PI_FILE_TRANSFER_BEGIN_MISSING	A uuencoded file transferred did not start with "begin" followed by the proper filename
-1051	PI_FILE_TRANSFER_ERROR_TEMP_FILE	Could not create/read file on host PC
-1052	PI_FILE_TRANSFER_CRC_ERROR	Checksum error when transferring a file to/from the controller
-1053	PI_COULDNT_FIND_PISTAGES_DAT	The PiStages.dat database could not be found. This file is required to connect a stage with the CST command
-1054	PI_NO_WAVE_RUNNING	No wave being output to specified axis
-1055	PI_INVALID_PASSWORD	Invalid password
-1056	PI_OPM_COM_ERROR	Error during communication with OPM (Optical Power Meter), maybe no OPM connected
-1057	PI_WAVE_EDITOR_WRONG_PARAMNUM	WaveEditor: Error during wave creation, incorrect number of parameters
-1058	PI_WAVE_EDITOR_FREQUENCY_OUT_OF_RANGE	WaveEditor: Frequency out of range
-1059	PI_WAVE_EDITOR_WRONG_IP_VALUE	WaveEditor: Error during wave creation, incorrect index for integer parameter
-1060	PI_WAVE_EDITOR_WRONG_DP_VALUE	WaveEditor: Error during wave creation, incorrect index for floating point parameter
-1061	PI_WAVE_EDITOR_WRONG_ITEM_VALUE	WaveEditor: Error during wave creation, could not calculate value
-1062	PI_WAVE_EDITOR_MISSING_GRAPH_COMPONENT	WaveEditor: Graph display component not installed
-1063	PI_EXT_PROFILE_UNALLOWED_CMD	User Profile Mode: Command is not allowed, check for required preparatory commands
-1064	PI_EXT_PROFILE_EXPECTING_MOTION_ERROR	User Profile Mode: First target position in User Profile is too far from current position

-1065	PI_EXT_PROFILE_ACTIVE	Controller is (already) in User Profile Mode
-1066	PI_EXT_PROFILE_INDEX_OUT_OF_RANGE	User Profile Mode: Block or Data Set index out of allowed range
-1067	PI_PROFILE_GENERATOR_NO_PROFILE	ProfileGenerator: No profile has been created yet
-1068	PI_PROFILE_GENERATOR_OUT_OF_LIMITS	ProfileGenerator: Generated profile exceeds limits of one or both axes
-1069	PI_PROFILE_GENERATOR_UNKNOWN_PARAMETER	ProfileGenerator: Unknown parameter ID in Set/Get Parameter command
-1070	PI_PROFILE_GENERATOR_PAR_OUT_OF_RANGE	ProfileGenerator: Parameter out of allowed range
-1071	PI_EXT_PROFILE_OUT_OF_MEMORY	User Profile Mode: Out of memory
-1072	PI_EXT_PROFILE_WRONG_CLUSTER	User Profile Mode: Cluster is not assigned to this axis
-1073	PI_UNKNOWN_CLUSTER_IDENTIFIER	Unknown cluster identifier
-1074	PI_INVALID_DEVICE_DRIVER_VERSION	The installed device driver doesn't match the required version. Please see the documentation to determine the required device driver version.
-1075	PI_INVALID_LIBRARY_VERSION	The library used doesn't match the required version. Please see the documentation to determine the required library version.
-1076	PI_INTERFACE_LOCKED	The interface is currently locked by another function. Please try again later.
-1077	PI_PARAM_DAT_FILE_INVALID_VERSION	Version of parameter DAT file does not match the required version. Current files are available at www.pi.ws.
-1078	PI_CANNOT_WRITE_TO_PARAM_DAT_FILE	Cannot write to parameter DAT file to store user defined stage type.
-1079	PI_CANNOT_CREATE_PARAM_DAT_FILE	Cannot create parameter DAT file to store user defined stage type.
-1080	PI_PARAM_DAT_FILE_INVALID_REVISION	Parameter DAT file does not have correct revision.
-1081	PI_USERSTAGES_DAT_FILE_INVALID_REVISION	User stages DAT file does not have correct revision.

13. Index