queensgate
a brand of PRIOR

User Manual

NANOSCAN NPC-D SERIES NANOMECHANISM CONTROLLER CONTROLLER INTERFACE LIBRARY

Application Programming Interface allowing configuration and control from customer software

Document No: EN-014217-UM Rev 2.0

Safety Precautions

WARNINGS

HAZARDOUS VOLTAGES

The Product generates high voltages and relies on the provision of a protective earth (ground) conductor to prevent user accessible components developing a hazardous potential in the event of an insulation failure. This protective earth is provided by the external power supply and only an approved power supply should be used with the product. Additional protection is provided by special NanoMechanism interface connectors and cable assembles. To maintain the integrity of the operator safety systems only approved NanoMechanisms and cables should be used with the product. The product should not be used if there are any signs of damage or if the equipment is believed to be faulty. It should be returned to the manufacturer for investigation and repair.

DO NOT remove the equipment's protect covers. There are no user serviceable parts within the equipment and removal of the cover will expose the user to potential high voltage hazards and will invalidate the Warranty.

Do read the manual before using the controller to understand how to correctly and safely operate the product. Incorrect use of the equipment may lead to personal injury or damage to property. Always turn the equipment off and remove the mains plug when not in use. Always use the equipment as specified in this manual.

CAUTIONS

ELECTROSTATIC SENSITIVE DEVICES (ESD)

The unit contains components that are susceptible to damage through electrostatic discharge at the NanoMechanism and interface connectors. Removal of protective connector covers and connection of cables should be performed in a static safe environment using approved static safety handling procedures.

ENVIRONMENT

The unit is designed for use in-doors in a dry environmentally controlled manufacturing facility, office or laboratory. The temperature and relative humidity should be kept within those specified in Table 2.1. Significant dust or acoustic/mechanical vibration may cause faulty operation or damage to components so should be avoided. Maintain adequate cooling of the controller by not restricting the air flow to and from the fan cooling vents. For prolonged periods of operation it is advisable to keep the environmental humidity to a minimum.

DIRECTIVE AND STANDARDS APPLIED:

CE

2004/108/EC	EMC Directive BS EN 61326-1:2006 Electrical equipment for measurement, control and laboratory use EMC requirements — Part 1: General requirements FCC part 15, subpart B
2006/95/EC	Low Voltage Directive BS EN 61010-1:2010 Safety requirements for electrical equipment for measurement, control, and laboratory use - Part 1: General requirements
2003/108/EC	WEEE Directive

Table of Contents

1 Introduction 5
1.1 Overview 5
2 Controller interface API conventions 6
2.1 DLL string handling conventions 6
2.2 Units and units types 6
2.3 Error reporting for invalid commands 7
2.4 Error reporting from the controller 7
2.5 Thread safety 7
2.6 Multiple controller interface sessions 7
3 Controller interface API 9
3.1 Reporting DLL version 9
3.2 Creating and destroying a controller interface session 9
3.3 Device discovery 9
3.4 Comms links available 10
3.4.1 Windows serial (RS-232 or USB emulated serial) port 10
3.4.2 Linux serial (RS-232) port 10
3.4.3 Linux USB emulated serial (gadget serial) port 10
3.4.4 Ethernet 11
3.4.5 Controller emulator 11
3.5 Opening and closing a comms link 11
3.6 Checking controller details 12
3.7 Requesting available commands 12
3.8 Requesting command details 13
3.8.1 Requesting command parameters 13
3.8.2 Requesting command return values 15
3.9 Commands 16

Related documents

Document Ref	Title	Usage
EN-002958-UM	NPC-D-5000 Series NanoMechanism Controller - Command Set And Control System	Lists and explains usage of NPC-D-5000 controller commands and structure of control system
EN-014429-UM	NanoScan NPC-D-6xx0 Series NanoMechanism Controller - Command Set And Control System	Lists and explains usage of NPC-D 6xx0 controller commands and structure of control system

Revision history

Revision	Changes
1.0	First version.
2.0	General: Reformatted for Prior Scientific and NanoScan rebranding. Front page: Corrected document number. Header: Updated document title. 2.5, “Thread safety”; 2.6, “Multiple controller interface sessions”: Divided the previous single section into two, and clarified interactions. 3.7, “Requesting available commands”: Added that commands available are also dependent on controller firmware version. 3.9, “Commands”: For controller interface library version 2.4.x onwards, specified how to send multiple commands and read back multiple results.

1 Introduction

1.1 Overview

Queensgate NanoMechanisms combine a flexure-guided mechanism, a piezo actuator and a precision capacitive position sensor to allow highly-accurate position measurement. These NanoMechanisms are generally referred to as “stages”. Where multi-axis control is required, multiple stages may be assembled together, or a single stage may be designed to provide control in two or more axes.

The NanoScan NPC-D series digital controllers provide closed-loop positioning control for one or more stage axes. The controller incorporates a high-voltage power amplifier to drive the piezo actuator, a precision measurement circuit to calculate the stage position, digital signal processing to provide closed-loop control of the stage(s), and various interfacing methods to set the commanded position and system calibration.

To allow straightforward user control of the system, the Controller Interface DLL provides a standard method for a PC running Windows or Linux to interact with the controller. The user need only tell the DLL how the PC is connecting to the controller, and the DLL handles all details of the communications link. Requests/commands are provided to the DLL as text strings; the DLL translates these into the appropriate messages to/from the controller over the comms link; and responses from the controller are returned by the DLL as text strings.

The same DLL is used for all controllers. The DLL automatically detects which controller type is connected (as well as what security privileges are enabled), and only displays the relevant commands for that controller. Commands for NPC-D-5000 series controllers are covered by manual EN-002958UM, and commands for NPC-D-6000 series controllers are covered by manual EN-014429-UM. This document will not go into further detail about controller functionality or commands, since this is controller-specific.

Commands are always the same regardless of the comms link used and regardless of whether the PC is running Windows or Linux. Most commands will also be the same for all digital controllers, since they share a common control loop architecture.

This removes the need for the user to handle any low-level interfacing to the controller within their application, and allows easy porting across different comms links (e.g. RS-232, USB or Ethernet) and different platforms. It also greatly minimises the work required to port applications to future digital controllers.

The DLL has a simple function-based application programmer interface (API), as a standard Windows “C” DLL or Linux shared object (SO) library. This allows the interface to be used by any Windows or Linux software. Applications importing the DLL may be written directly in any programming language such as C, C++ or Python, or may be designed in environments such as Matlab or Labview.

2 Controller interface API conventions

2.1 DLL string handling conventions

The following conventions are followed by the DLL when reading/writing strings.

Where the DLL is required to read a string provided by the user, the string must be in UTF-8 format and stored in a null-terminated byte array.

Where the DLL is required to report a string to the user, the user must provide a byte array into which the string will be copied, as well as the length of the array in bytes. The DLL will copy the string into this array in UTF-8 format, and will null-terminate the string. Functions which report a string to the user will return the length of the string in bytes (not in UTF-8 characters).

If the user provides a byte array which is shorter than the string to be copied, the DLL will copy the string into this array up to the size of the array, and will null-terminate the string at the last available character space. The function will return the number of bytes required to store the entire string (including null-terminator).

If the user wishes to dynamically allocate the byte array, functions may be called with a string length of zero or the pointer to the string set to NULL. The function will return the number of bytes required to store the entire string (including null-terminator).

All strings returned by the DLL are in English. There is no provision for internationalisation within the DLL, because the intended users are engineers who are expected to be familiar with documentation and coding in English. In the rare cases where strings returned by the DLL may be shown on a user interface, the application must provide any translation required.

2.2 Units and unit's types

For all commands, it is possible to request the units for each parameter and returned value.

Where units may not always be the same (e.g. distance may be reported in picometres for linear stages or picoradians for angular stages), this allows the application to check the units for a parameter or returned value so that they can be reported correctly.

Each parameter's units are further identified as a "units type", for example "distance" or "time". It is envisaged that in future the DLL will allow different units to be selected for a units type, so that all commands with parameters or return values of that units type will have the appropriate scaling applied. For example, the units type "distance" could be changed to "mil" if it is more appropriate for distance to be reported in thousandths of an inch.

For consistency with linear/angular stages, and for forwards compatibility, applications should always check the units used for a parameter or return value.

2.3 Error reporting for invalid commands

If a command is not recognised (perhaps due to a typo, or perhaps because security prevents that command being used), the function `DoCommand()` will return a negative value. No command will be issued to the controller.

2.4 Error reporting from the controller

If a command is issued to the controller, and the controller reports an error, the function `DoCommand()` will return success. Instead of the expected return value(s) for the command, the command will always report two return values.

The first value will always be named “error” and the value will always be a string “FAILED”.

The second value will always be named “errcode” and the value will be a string reporting the reason why the command failed.

It is highly recommended when running commands that the first return value is checked to ensure the return value name is not “error”. If it is, the application should take any appropriate mitigating action. Depending on the command, the second return value “errcode” may allow the application to take more appropriate action for the specific error. These errors are covered in more detail in the command set manual.

2.5 Thread safety

The DLL is thread-safe, so it may safely be used by several threads consecutively. However the majority of functions are atomic, so actions by one thread may cause another thread to block until the first thread has completed.

The functions `GetResult()` and `GetResultName()` always return the results of the latest command issued by `DoCommand()` for their controller interface session. As a result, it is strongly recommended that only one thread calls `DoCommand()` to issue commands for each controller interface session, or if not, that some suitable resource locking mechanism is used.

2.6 Multiple controller interface sessions

It is possible to communicate with the controller on multiple controller interfaces simultaneously. For example, if the controller is fitted with RS-232 and USB then both may be used, with one controller interface session each. If an Ethernet connection is used then multiple controller interface sessions may be opened on the same static IP address. These may be on separate PCs, or may be on the same PC and will still be independent of each other. The same PC may likewise communicate with any number of controllers simultaneously.

Where multiple controller interfaces are used simultaneously, it is strongly recommended that one session is allocated control, able to change system state, and the other sessions are limited to being “observers” only. This may be enforced by removing security permissions from “observer” sessions, so that commands which would change system state are not permitted. If this pattern is not followed, care must be taken to ensure one session does not inadvertently

change or invalidate settings configured by another session. This may cause unexpected system behaviour.

A controller is only able to process one command at a time. Commands are processed in strict order of arrival, so commands from one controller interface will delay processing from other controller interfaces. Where multiple controller interfaces are used simultaneously, care must be taken to ensure the overall command throughput is acceptable, and that “observer” sessions reading the system state do not prevent control commands from being processed in a timely manner.

3 Controller interface API

3.1 Reporting DLL version

An application may need to check or report which DLL version is in use.

`void` **`GetDllVersion`**`(int*` *`majorVersion`*`, int*` *`minorVersion`*`, int*` *`buildNumber`*`)`

Reports the version number of this DLL.

`majorVersion`, *`minorVersion`* and *`buildNumber`* pass back the version number of this DLL.

3.2 Creating and destroying a controller interface session

An application may need to talk to several controllers over different interfaces. Each controller interface session must use the functions below to get a unique handle for that session, and to clean up that session when it is no longer required.

`ControllerInterfaceHandle` **`Init`**`(void)`

Prepares a new controller interface session for use, and returns a handle to this session. Each session has a unique handle, and all calls to other functions must use this handle to identify the session in use. The function returns NULL if a new handle cannot be allocated.

`void` **`Uninit`**`(ControllerInterfaceHandle` *`handle`*`)`

Deletes the session for the session handle *`handle`*. The handle may not be used after this function is called.

If the session has an open connection to a controller, `CloseSession()` will automatically be called so that the comms link is not left locked.

3.3 Device discovery

If the comms link to be used is not known, an application may need to locate available devices on available comms links. This is known as “device discovery”.

`int` **`FindDevices`**`(ControllerInterfaceHandle` *`handle`*`)`

Initiates searching for devices on all available comms links. The function does not return until all available comms links have been checked, which may take a few seconds.

`handle` specifies the controller interface session being used for discovery. This session must not be currently connected to a comms link.

Returns number of discovered devices. If the value is negative then discovery failed.

`int` **`GetDevice`**`(ControllerInterfaceHandle` *`handle`*`, const int` *`index`*`, char * buffer, const int bufferLen)`

Retrieves the comms link for a device discovered in the last call to `FindDevices()`. The list of discovered devices will remain unchanged for this session until `FindDevices()` is called again. Calls to `FindDevices()` for other sessions will not change the list of discovered devices for this session.

`handle` specifies the controller interface session being used for discovery. This session must not be currently connected to a comms link.

`index` specifies the index of the device in the discovered device list (where the first item in the list is index zero).

`buffer` is a pointer to a byte array of length *`bufferLen`* bytes. The comms link identifier is copied into this, up to the length of the array.

Returns the number of bytes in *`buffer`* used for the comms link identifier, or the size of the required buffer if the buffer is too small (or zero length). If the value is negative then *`index`* is beyond the end of the discovered device list.

NOTE: Device discovery is currently not implemented. Functions exist, but FindDevices() always returns zero and GetDevice() always returns negative.

3.4 Comms links available

3.4.1 Windows serial (RS-232 or USB emulated serial) port

On Windows, an RS-232 port is specified as “COMx”, where “x” is a number from 1 to 255 (e.g. “COM123”). The USB connection emulates an RS-232 port, so this is similarly specified as “COMx”. Use the Windows Device Manager to identify how these correspond to a physical RS-232 port on the PC or to a USB connection.

3.4.2 Linux serial (RS-232) port

On Linux, an RS-232 port is specified as “/dev/ttySx”, where “x” is a number from 0 onwards (e.g. “/dev/ttyS3”). Check your Linux distribution for how to identify the relevant TTY corresponding to a physical RS-232 port on the PC.

3.4.3 Linux USB emulated serial (gadget serial) port

On Linux, the USB connection again emulates an RS-232 port, using the “gadget serial” interface. This is specified as “/dev/ttyGSx”, where “x” is a number from 0 onwards (e.g. “/dev/ttyGS4”). Check your Linux distribution for how to manage USB devices and how to identify the relevant TTY corresponding to a USB connection.

3.4.4 Ethernet

For both Windows and Linux, a connection over Ethernet is specified as the static IP address of the controller (e.g. “192.168.0.7”). See the command set manual for details of the commands to set/get the controller’s static IP address.

NOTE: Control over Ethernet is not currently implemented.

3.4.5 Controller emulator

For testing purposes, the DLL can emulate a controller when no controller is fitted. All commands will return zero, but this may be sufficient to test that an application runs.

The simulator is specified as “sim:/NPCxxxx”, where “xxxx” is the controller part number (for example, “sim:/NPC6330” to emulate an NPC-6330 controller). Different controllers have slightly different command sets, so this allows the DLL to select the relevant commands for emulation.

Where different versions of controller firmware have slightly different commands sets available, the simulator may further be specified as “sim:/NPCxxxx/a.b.c”, where “a.b.c” is the controller firmware version number. The DLL will select only commands which are applicable for that firmware version.

3.5 Opening and closing a comms link

To talk to a device, it is necessary to open a connection to the device for that comms link.

Where a comms link can only be used by one application at a time (for example a serial port), only one session can be open at a time for that comms link. Further attempts to open a session for that comms link will fail. When the session is closed, the comms link will again be available for other applications to use. Where a comms link may be used by multiple applications (for example an IP address over Ethernet), multiple sessions may connect to that comms link.

A session may only have one comms link open at once. Attempts to open another comms link when the session already has a comms link open will fail. After `CloseSession()` is called, the same session handle may be used to open another comms link.

`int` **`OpenSession`**`(ControllerInterfaceHandle` *`handle`*`, const char *` *`device`*`)`

Opens a comms link connection to a device for this session.

`handle` specifies the controller interface session. This session must not be currently connected to a comms link.

`device` is a pointer to a null-terminated byte array containing the comms link identifier. This specifies the device to be opened, e.g. a serial port or an IP address.

Returns positive if the session has been opened, or zero if the device could not be connected to or the session already has a comms link connection open.

`void` **`CloseSession`**`(ControllerInterfaceHandle` *`handle`*`)`

Closes the comms link connection for this session. The current device is disconnected, and the comms link and controller interface session are free for reuse. If the session does not have a comms link open, this function has no effect.

`handle` specifies the controller interface session.

3.6 Checking controller details

The majority of controller details are best read by the relevant commands, with one exception.

`int` **`GetChannels`**`(ControllerInterfaceHandle` *`handle`*`)`

Reports the number of stage channels supported by the controller connected for this session.

`handle` specifies the controller interface session.

Returns the number of stage channels, or negative if no controller is connected for this session.

3.7 Requesting available commands

To use a device, the available commands may need to be obtained. The application may request all commands, or may request only commands starting with a specific string.

The commands available will vary depending on the security level currently unlocked for this session's comms link connection. Commands exist to unlock different security levels, as appropriate for the current user. Where commands are not permitted for the current security level, they will not be shown in this list.

The commands available will also vary depending on the controller connected, and on the firmware version. Many commands will be common to all controllers, but some commands will be controller-specific. Where features were added at a particular firmware versions, commands to support those features will not be available for controllers using previous versions of firmware.

`int` **`FindCommands`**`(ControllerInterfaceHandle` *`handle`*`, const char *` *`filter`*`)`

Retrieves the list of available commands.

`handle` specifies the controller interface session.

`filter` is a pointer to a null-terminated byte array. If this is NULL, a list of all available commands will be reported. If this is not NULL, a list of all available commands beginning with this string will be reported.

Returns number of commands available. Some commands will always be available, regardless of the controller or security level. If the value is negative then there is no open session.

`int` **`GetCommand`**`(ControllerInterfaceHandle` *`handle`*`, const int` *`index`*`, char * buffer, const int bufferLen)`

Gets a command from the list retrieved in the last call to `FindCommands()` for this session.

`handle` specifies the controller interface session.

`index` specifies the index of the command in the list (where the first item in the list is index zero).

`buffer` is a pointer to a byte array of length *`bufferLen`* bytes. The command name is copied into this, up to the length of the array.

Returns the number of bytes in *`buffer`* used for the command name, or the size of the required buffer if the buffer is too small (or zero length). If the value is negative then *`index`* is beyond the end of the list.

3.8 Requesting command details

For each supported command, the application may require further details of the command. In particular, the application should check what units are used for parameters passed to the command and values returned, so that scalings can be applied appropriately.

`int` **`GetCommandDescription`**`(ControllerInterfaceHandle` *`handle`*`, const char* commandName, char* buffer, const int bufferLen)`

Gets a brief description of the command.

`handle` specifies the controller interface session.

`commandName` is a pointer to a null-terminated byte array containing the command for which the description is to be reported.

`buffer` is a pointer to a byte array of length *`bufferLen`* bytes. The command description is copied into this, up to the length of the array.

Returns the number of bytes in *`buffer`* used for the command description, or the size of the required buffer if the buffer is too small (or zero length). If the value is negative then *`commandName`* does not contain the name of a valid command.

3.8.1 Requesting command parameters

`int` **`GetCommandParameters`**`(ControllerInterfaceHandle` *`handle`*`, const char* commandName)`

Gets the number of parameters to be passed to this command.

`handle` specifies the controller interface session.

`commandName` is a pointer to a null-terminated byte array containing the command for which parameter details are to be reported.

Returns the number of parameters to be passed for this command. This may be zero, for commands which do not take any parameters. If the value is negative then *commandName* does not contain the name of a valid command.

`int` **`GetCommandParameterName`**`(ControllerInterfaceHandle` *`handle`*`, const char*` *`commandName`*`, const int` *`paramIndex`*`, char*` *`buffer`*`, const int` *`bufferLen`*`)`

Gets the name of one parameter to be passed to this command.

handle specifies the controller interface session.

commandName is a pointer to a null-terminated byte array containing the command for which the parameter is to be reported.

paramIndex is the index of the parameter requested (where the first parameter is index zero).

buffer is a pointer to a byte array of length *bufferLen* bytes. The parameter name is copied into this, up to the length of the array.

Returns the number of bytes in *buffer* used for the parameter name, or the size of the required buffer if the buffer is too small (or zero length). If the value is negative then *commandName* does not contain the name of a valid command, or *paramIndex* is beyond the number of parameters.

`int` **`GetCommandParameterUnitsType`**`(ControllerInterfaceHandle` *`handle`*`, const char*` *`commandName`*`, const int` *`paramIndex`*`, char*` *`buffer`*`, const int` *`bufferLen`*`)`

Gets the units type (e.g. “distance”) for one parameter to be passed to this command.

handle specifies the controller interface session.

commandName is a pointer to a null-terminated byte array containing the command for which the parameter units type is to be reported.

paramIndex is the index of the parameter requested (where the first parameter is index zero).

buffer is a pointer to a byte array of length *bufferLen* bytes. The parameter units type is copied into this, up to the length of the array.

Returns the number of bytes in *buffer* used for the parameter units type, or the size of the required buffer if the buffer is too small (or zero length). If the value is negative then *commandName* does not contain the name of a valid command, or *paramIndex* is beyond the number of parameters.

`int` **`GetCommandParameterUnits`**`(ControllerInterfaceHandle` *`handle`*`, const char*` *`commandName`*`, const int` *`paramIndex`*`, char*` *`buffer`*`, const int` *`bufferLen`*`)`

Gets the units (e.g. “pm”) for one parameter to be passed to this command.

handle specifies the controller interface session.

commandName is a pointer to a null-terminated byte array containing the command for which the parameter units are to be reported.

paramIndex is the index of the parameter requested (where the first parameter is index zero).

buffer is a pointer to a byte array of length *bufferLen* bytes. The parameter units are copied into this, up to the length of the array.

Returns the number of bytes in *buffer* used for the parameter units, or the size of the required buffer if the buffer is too small (or zero length). If the value is negative then *commandName* does not contain the name of a valid command, or *paramIndex* is beyond the number of parameters.

3.8.2 Requesting command return values

`int` **`GetCommandResults`**`(ControllerInterfaceHandle` *`handle`*`, const char*` *`commandName`*`)`

Gets the number of values returned from this command if the command is successful.

handle specifies the controller interface session.

commandName is a pointer to a null-terminated byte array containing the command for which return value details are to be reported.

Returns the number of return values from this command. All commands will return at least one value, even if this is only a dummy value to indicate success. If the value is negative then *commandName* does not contain the name of a valid command.

`int` **`GetCommandResultName`**`(ControllerInterfaceHandle` *`handle`*`, const char*` *`commandName`*`, const int` *`resultIndex`*`, char*` *`buffer`*`, const int` *`bufferLen`*`)`

Gets the name of one result returned from this command if the command is successful.

handle specifies the controller interface session.

commandName is a pointer to a null-terminated byte array containing the command for which the result name is to be reported.

resultIndex is the index of the result requested (where the first result is index zero).

buffer is a pointer to a byte array of length *bufferLen* bytes. The result name is copied into this, up to the length of the array.

Returns the number of bytes in *buffer* used for the result name, or the size of the required buffer if the buffer is too small (or zero length). If the value is negative then *commandName* does not contain the name of a valid command, or *resultIndex* is beyond the number of results.

`int` **`GetCommandResultUnitsType`**`(ControllerInterfaceHandle` *`handle`*`, const char*` *`commandName`*`, const int` *`resultIndex`*`, char*` *`buffer`*`, const int` *`bufferLen`*`)`

Gets the units type (e.g. “distance”) for one result returned from this command if the command is successful.

`handle` specifies the controller interface session.

`commandName` is a pointer to a null-terminated byte array containing the command for which the result units type is to be reported.

`resultIndex` is the index of the result requested (where the first result is index zero).

`buffer` is a pointer to a byte array of length *`bufferLen`* bytes. The result units type is copied into this, up to the length of the array.

Returns the number of bytes in *`buffer`* used for the result units type, or the size of the required buffer if the buffer is too small (or zero length). If the value is negative then *`commandName`* does not contain the name of a valid command, or *`resultIndex`* is beyond the number of results.

`int` **`GetCommandResultUnits`**`(ControllerInterfaceHandle` *`handle`*`, const char*` *`commandName`*`, const int` *`resultIndex`*`, char*` *`buffer`*`, const int` *`bufferLen`*`)`

Gets the units (e.g. “pm”) for one result returned from this command.

`handle` specifies the controller interface session.

`commandName` is a pointer to a null-terminated byte array containing the command for which the result units are to be reported.

`paramIndex` is the index of the result requested (where the first result is index zero).

`buffer` is a pointer to a byte array of length *`bufferLen`* bytes. The result units are copied into this, up to the length of the array.

Returns the number of bytes in *`buffer`* used for the result units, or the size of the required buffer if the buffer is too small (or zero length). If the value is negative then *`commandName`* does not contain the name of a valid command, or *`resultIndex`* is beyond the number of results.

3.9 Commands

Once a session has been opened, commands from this list can be issued.

To allow greater portability and ease of future expansion, a command is formatted as a single string containing the command name and all parameters required, which is passed to the **`DoCommand()`** function . The DLL sends and receives messages to and from the controller, as appropriate for the command. A command will then return one or more results, where each result is formatted as a string containing the result name and a string containing the result value. These are read back using the **`GetResultName()`** and **`GetResult()`** functions.

As of controller interface DLL version 2.4.1, it is possible to pass multiple commands to the **DoCommand()** function, separated by a return character (ASCII code 13, represented by "\r" in many languages) or newline character (ASCII code 10, represented by "\n" in many languages). The DLL and controller are able to combine multiple commands into a single comms link message, allowing more efficient processing of commands with greatly-reduced overheads. This can give up to 10x improvement in command throughput compared to calling the **DoCommand()** function separately for each command.

When sending multiple commands, results are provided in the order in which commands are issued. The DLL does not report which results are linked to which command, so the user's application is responsible for appropriate processing of results, knowing the order in which commands were issued.

For further efficiency, as of version 2.4.1 the DLL also provides **GetAllResultNames()** and **GetAllResults()** functions to report all results in a single function call. Result names and values are followed by newline character (ASCII code 10, represented by "\n" in many languages, apart from the last which is null-terminated.

The **DoCommand()** function returns when the command or commands have been processed, which will typically be fast for a single command. When sending multiple commands, care must be taken that the calling application is not prevented from responding to user input whilst the **DoCommand()** function runs. There is also no mechanism for aborting the **DoCommand()** function, so again care must be taken that too many commands are not requested at once if the user may wish to halt the operation (for example when reading a large snapshot). Because of these factors, it is recommended that users should not send more than 20 commands at once. Beyond this there is typically little or no improvement in command throughput, and the time to process 20 commands is generally low enough that application responsiveness is unaffected.

Earlier DLL versions (before version 2.4.1) only support passing a single command to the **DoCommand()** function and do not provide the provides **GetAllResultNames()** and **GetAllResults()** functions. For earlier DLL versions, passing multiple commands will cause **DoCommand()** to report an error. Attempting to get pointers to the functions dynamically (using **GetProcAddress()** in Windows or **dlsym()** in Linux) will fail, and attempting to link to the **GetAllResultNames()** and **GetAllResults()** functions will cause a compile error or runtime error depending on when the older DLL is used. Applications which must be backwardly-compatible with 2.2.x releases of the DLL must not use the multiple-command syntax for **DoCommand()** and must not attempt to call the **GetAllResultNames()** and **GetAllResults()** functions.

Older controller firmware may not be able to combine multiple commands into a single comms link message. The DLL is able to detect this and send appropriate messages over the comms link. This happens transparently, so the user does not need to be aware of this.

Note that DLL commands themselves are backwardly-compatible, and newer versions of the DLL will never remove earlier commands or remove support for older controller firmware. Users therefore are recommended to keep the DLL version up-to-date, upgrading as necessary. If developing new software, it is recommended that the latest DLL versions should always be used.

`int` **`DoCommand`**`(ControllerInterfaceHandle` *`handle`*`, const char*` *`command)`*

Issues command(s) to the controller.

`handle` specifies the controller interface session.

`command` is a pointer to a null-terminated byte array containing the command(s) and all parameters.

Returns the number of return values from the command(s), or negative in case of error. If the value is negative then *`command`* does not contain the name of a valid command, not enough parameters were specified for a command, or the comms link failed to send/receive a command.

`int` **`GetResultName`**`(ControllerInterfaceHandle` *`handle`*`, const int` *`rindex`*`, char*` *`buffer`*`, const int` *`bufferLen`*`)`

Gets the name of one result returned from the latest command(s) sent.

`handle` specifies the controller interface session.

`index` is the index of the result requested (where the first result is index zero).

`buffer` is a pointer to a byte array of length *`bufferLen`* bytes. The result name is copied into this, up to the length of the array.

Returns the number of bytes in *`buffer`* used for the result name, or the size of the required buffer if the buffer is too small (or zero length). If the value is negative then *`index`* is beyond the number of results.

`int` **`GetAllResultNames`**`(ControllerInterfaceHandle` *`handle`*`, const int` *`rindex`*`, char*` *`buffer`*`, const int` *`bufferLen`*`)`

Gets the names of all results returned from the latest command(s) sent.

`handle` specifies the controller interface session.

`buffer` is a pointer to a byte array of length *`bufferLen`* bytes. The result names are copied into this, up to the length of the array.

Returns the number of bytes in *`buffer`* used for the result names, or the size of the required buffer if the buffer is too small (or zero length).

`int` **`GetResult`**`(ControllerInterfaceHandle` *`handle`*`, const int` *`index`*`, char*` *`buffer`*`, const int` *`bufferLen`*`)`

Gets the value for one result returned from the latest command(s) sent.

`handle` specifies the controller interface session.

`index` is the index of the result requested (where the first result is index zero).

`buffer` is a pointer to a byte array of length *`bufferLen`* bytes. The result value is copied into this, up to the length of the array.

Returns the number of bytes in *`buffer`* used for the result value, or the size of the required buffer if the buffer is too small (or zero length). If the value is negative then *`index`* is beyond the number of results.

`int` **`GetAllResults`**`(ControllerInterfaceHandle` *`handle`*`, char*` *`buffer`*`, const int` *`bufferLen`*`)`

Gets the values for all results returned from the latest command(s) sent.

`handle` specifies the controller interface session.

`buffer` is a pointer to a byte array of length *`bufferLen`* bytes. The result values are copied into this, up to the length of the array.

Returns the number of bytes in *`buffer`* used for the result values, or the size of the required buffer if the buffer is too small (or zero length).