DCX-PCI100

Modular Multi-Axis Motion Control System

User's Manual

Revision 1.0c

Precision MicroControl Corporation
2075-N Corte del Nogal
Carlsbad, CA 92009-1415 USA

Tel: (760) 930-0101
Fax: (760) 930-0222

www.pmccorp.com

Information: info@pmccorp.com
Technical Support: support@pmccorp.com

LIMITED WARRANTY

All products manufactured by PRECISION MICROCONTROL CORPORATION are guaranteed to be free from defects in material and workmanship, for a period of five years from the date of shipment. Liability is limited to FOB Factory repair, or replacement, of the product. Other products supplied as part of the system carry the warranty of the manufacturer.

PRECISION MICROCONTROL CORPORATION does not assume any liability for improper use or installation or consequential damage.

(c)Copyright Precision Micro Control Corporation, 1994-2004. All rights reserved.

Information in this document is subject to change without notice.

IBM and IBM-AT are registered trademarks of International Business Machines Corporation.
Intel and is a registered trademark of Intel Corporation.
Microsoft, MS-DOS, and Windows are registered trademarks of Microsoft Corporation.
Acrobat and Acrobat Reader are registered trademarks of Adobe Corporation.

Precision MicroControl
2075-N Corte del Nogal
Carlsbad, CA 92009-1415

Phone: (760)930-0101
Fax: (760)930-0222
World Wide Web: www.pmccorp.com
Email:
Information: info@pmccorp.com
Technical support: support@pmccorp.com
Sales: sales@pmccorp.com

Table of Contents

User manual revision history

Revision	*Date*	*Description*
1.0Pre	10/8/2001	Preliminary release
	1/18/2002	Added Amplifier Fault axis shut down options
1.0	5/8/2002	Release 1.0
1.0b	8/16/2002	Fixed DCX-MC110 J3 connector pin #2 description
		Edited DCX-MC110 jumper descriptions
		Edited DCX-MC110 graphics (pages 376 & 377)
		Edited DCX-MC100 graphic (page 373)
		Fixed DCX-MC400 connector pinout (all channels labeled as #1)
		Updated ribbon cable connector manufacturer part number
1.0c	12/16/2003	Updated installation instructions for MCAPI 3.4.1 or later
	12/17/2003	Added Troubleshooting flowcharts to the Appendix
	12/19/2003	Edited 'Setting Following Error' description (disabled by default)

Contact us at:

Precision MicroControl
2075-N Corte del Nogal
Carlsbad, CA 92009-1415

Phone: (760)930-0101
Fax: (760)930-0222
World Wide Web: www.pmccorp.com
Email:
Information: info@pmccorp.com
Technical support: support@pmccorp.com
Sales: sales@pmccorp.com

Chapter 1

Introduction

Motion controller - a device that uses a digital processor to coordinate the movement of mechanical systems.

The DCX-PCI100 is an Intel compatible PC computer based servo motor and I/O controller.

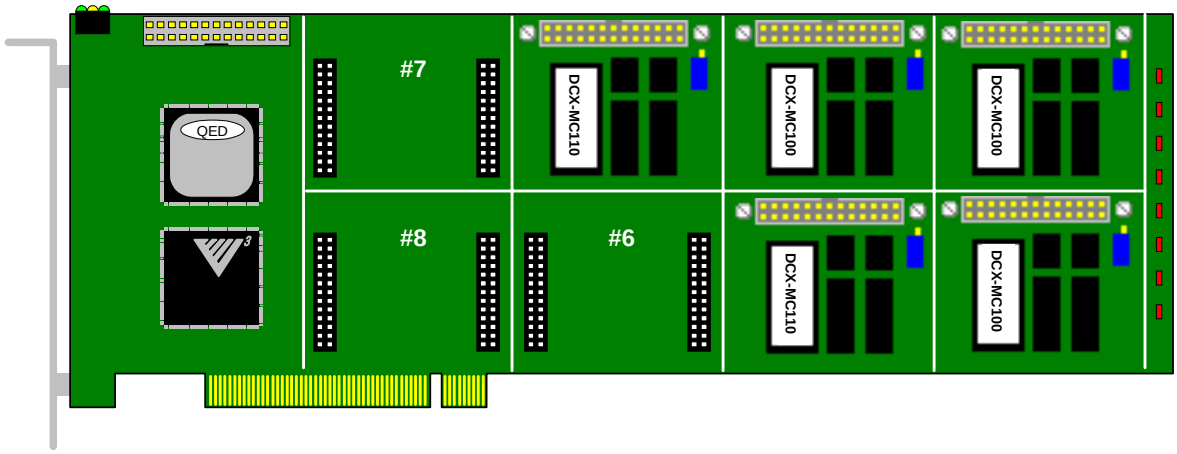


Figure 1: A DCX-PCI100 Motion Controller configured for 5 axes of servo motor control

In Windows 2000/Me/98 systems the DCX-PCI 100 is a true PCI 'plug and play' card. When the PC is turned on, the DCX-PCI100 is *dynamically addressed* into the memory map of the PC. The PC communicates with the motion controller via dual ported memory on the DCX-PCI100. The PC can issue commands (move a motor, change the velocity, etc.) to the controller, and retrieve data from the controller (report to position of an axis, report the state of a digital input, etc.) without interrupting the basic operations of the controller.

But a hardware based motion control card provides only one half of the overall motion control solution. State of the art motion control systems typically require sophisticated multi-threaded application programs and eye catching operator interfaces. PMC's **Motion Control Application Programming**

Interface (**MCAPI**) provides the machine designer with device drivers and a powerful function library for Windows 2000/NT/Me/98 based applications.

Figure 2: PMC's Windows Motion Control Panel

```
MCEnableAxis( HCTRLR hCtlr, Word xAxis, short int bState );
MCMoveRelative( HCTRLR hCtlr, Word xAxis, double Distance );
MCIsStopped( HCTRLR hCtlr, Word xAxis, double Timeout );
```

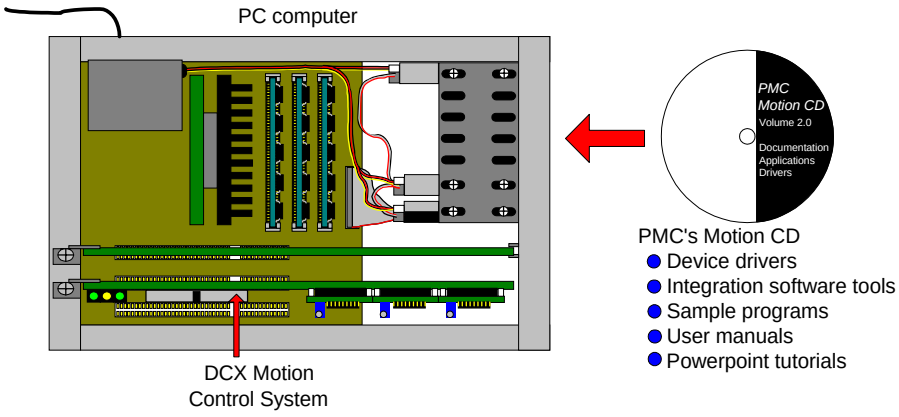
Figure 3: Function Library examples

The MCAPI supports today's popular programming environments including:

- C/C++
- Visual Basic
- Delphi
- LabVIEW

The DCX-PCI100 Motion Controller can be installed in most any Windows PC computer. It executes motion functions independent of the host, so other than the minimum requirements for the selected operating environment (2000/NT/ME/98), the DCX-PCI100 **does not require or use any additional PC resources** (CPU speed, PC memory, hard disk space, etc...).

All documentation, tutorials, and software (drivers, function library, diagnostics and utilities) are available on PMC's **MotionCD**.



The Modular DCX System

The modular architecture of the DCX system allows the user to '**mix and match**' DCX components to meet the **specific requirements of each application**. The DCX system controls the motion of as many as eight servo motors simultaneously. In addition the DCX modular system supports expandable digital I/O and analog I/O.

The term **DCX** refers to a system consisting of from 1 to 9 circuit boards assembled together to form a motion control assembly. The platform of a DCX system is the DCX-PCI100 "motherboard". It is a 'full' size (approximately 4" x 12.25") PCI peripheral card. It communicates with the PC host via the PCI bus. On board dual ported memory is used to pass motion commands and report data between the DCX controller and the PC. The on board CPU (**192MHz MIPS**) allows the DCX to operate autonomously from the PC, freeing the host to process critical events while the DCX handles all motion control. But please note - the DCX-PCI100 motherboard is the processing / communication / synchronizing engine of the DCX system, but on its own it provides no actual motion control.

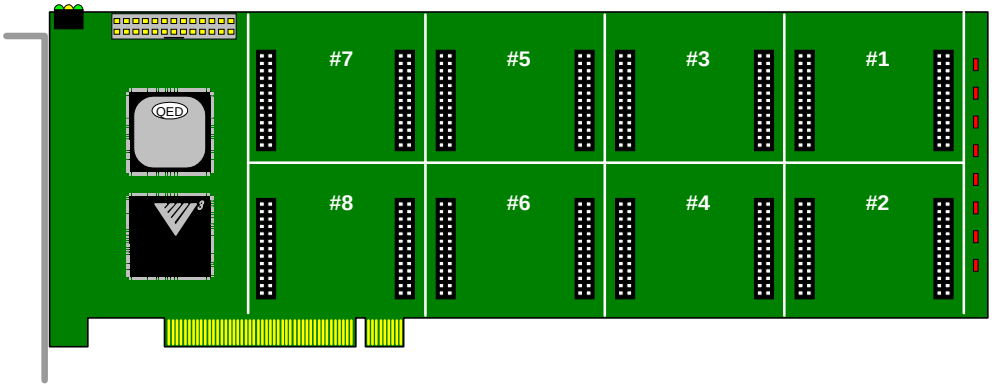


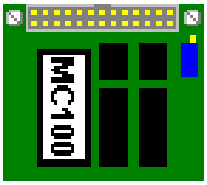
Figure 4: The DCX-PCI100 Motion Control Motherboard

To complete the DCX Modular Motion Control System, on to the DCX-PCI100 motherboard the user installs as many as eight, 2 inch square "daughter boards" known as "DCX modules" . DCX motion control modules provide:

- The motion control command output (DCX-MC100 = +/- 10V for servo amplifier, DCX-MC110 = 0.5A direct motor drive)
- PID filter (servo modules only)
- Trajectory Generator providing Trapezoidal Velocity Profiles (common accel / decel)
- Monitoring of TTL level axis I/O (+/- Limits, Home, Amp/Driver enable)
- Encoder interface and decode

The DCX-PCI100 motherboard currently supports four DCX modules, two for motion control and two for general purpose I/O. A key feature of the DCX system is its ability to sense which DCX modules are present. This results in easy system configuration; simply install whatever modules the application calls for. The logic on the motherboard will adjust its' operation accordingly.

DCX Motion Control Modules

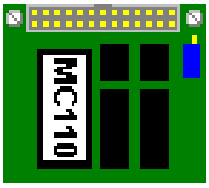


DCX-MC100 Servo Motor Control Module (to be used in conjunction with an external servo amplifier

Supported motor type: DC Brushless, Brush, Hydraulic Servo Valves, Pneumatic Servo Valves

Command output: **+/- 10 volt, 12 bit analog for use with servo amplifier**

I/O
Inputs, TTL (0 - +5V, low active), Encoder Coarse Home, Limit +, Limit -, and Amplifier Fault
Output TTL (0 - +5V, low active, 10ma max.) – Amplifier Inhibit
Feedback: Quadrature Incremental Encoder , 750 KHz maximum frequency,
Differential (A+, A-, B+, B-, Z-) or Single ended (A, B, Z-)



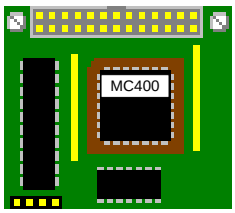
DCX-MC110 Servo Motor Control Module (for direct drive of small brush motors)

Supported motor type: Small DC Brush

Command output: **+/- 12 volt, 8 bit, 0.5A max.**

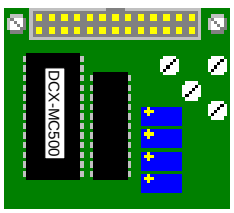
I/O
Inputs, TTL (0 - +5V, low active), Encoder Coarse Home, Limit +, Limit -, and Amplifier Fault
Output TTL (0 - +5V, low active, 10ma max.) – Amplifier Inhibit
Feedback: Quadrature Incremental Encoder, 750 KHz maximum frequency,
Differential (A+, A-, B+, B-, Z-) or Single ended (A, B, Z-)

DCX General Purpose I/O Modules



DCX-MC400 - **16 Channel Digital I/O Expansion module**

Each channel is individually programmable as either an input or output
TTL level (0 – 5 volt, 2 ma sink/source)

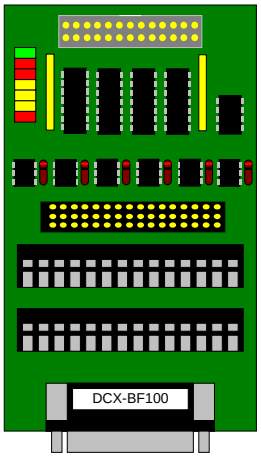


DCX-MC500 – **4 Channel Analog I/O Expansion module**

Inputs – 4 channels, 0 – 5 volts, 12 bit
Outputs – 4 channels, 0 – 5 volts and/or –10 - +10 volts, 12 bit

Ordering Options:
MC510 – 4 input channels only
MC520 – 4 output channels only

DCX Motion Control Breakout Assemblies

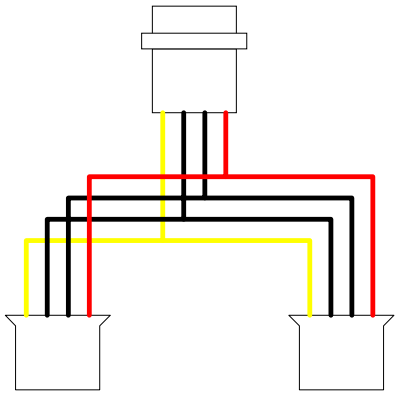


DCX-BF100 – Opto isolation and Interconnect assembly for DCX Servo Motor Control Modules (DCX-MC100, DCX-MC110)

Opto isolated inputs – Enc. Coarse Home, Limit +, Limit -, Amp Fault
Open collector output – Amplifier Enable
Differential receiver for Index +, Index –
External system connections via DB25 or two 14 contact screw terminal strip

LED indicators for:
- Amplifier enable
- Encoder Coarse Home
- Limit +
- Limit –
- Amplifier Fault

DCX Motion Control Accessories



Disk Drive Power Splitter Cable (P/N 71.060.A) – Connects PC computer +12 volts to the DCX-PCI100 motion controller

Chapter Contents

Chapter 2

Software and Controller Installation

The DCX-PCI100 is installed in a PCI slot of a PC computer or the passive back plane of an industrial computer. Power (+5V, +12V, and –12V), Ground reference, and communication (Address, Data, and Read/Write control signals) are supplied via the PCI edge connector. The DCX-PCI100 motion controller supports Windows 2000/NT/ME/98 operating systems, **the DCX-PCI100 does not support Windows 95 or 3.X**. .

DCX-PCI100 Motion Control System Installation

The basic steps for a **new** installation of the DCX-PCI100 motion controller for Windows 'Plug & Play' based applications are as follows:

The Microsoft convention for 'plug & play' devices is that the **drivers must be installed before the hardware**. For 'plug & play' operating systems (XP/2000/98) you must install MCAPI (3.4.1 or higher) **before** installing the DCX-PCI100 controller and 'booting' the computer.

- Turn on the computer and allow Windows to load completely
- Install PMC's motion control software (MCAPI 3.4.1 or higher) from the **MotionCD** or from PMC's web site **www.pmccorp.com**
- Exit from Windows and then turn off the computer
- With the computer power turned off, install the DCX-PCI100 motion control motherboard into an available PCI slot in the computer motherboard
- Turn on the computer, during the loading of Windows (except for NT4) the operating system should recognize that a new PCI card has been installed and the appropriate drivers will be selected
- The motion controller is now ready for testing

Installing the DCX Software (MCAPI)

DCX controllers ship with PMC's MotionCD, which includes the Motion Control API software. For the **most recent version** of the MCAPI please check the support page of PMC's website **www.pmccorp.com**

Downloading the Most Recent release of the Motion Control API from PMC's web site
Due to the dated nature of a CD, it is recommended that the user check PMC's web (**www.pmccorp.com**) site for the most recent release of the MCAPI. Go to the support page and select the link to the Motion Control API page.

Selecting the Motion Control API will begin the file download of this self extracting zip file. As shown in the following graphic, it is recommended that the file be saved to disk.

The installation of the MCAPI will begin upon launching the downloaded file. Follow the on screen instructions.

Installation from PMC's Motion CD
To install the Motion Control API software which includes: device drivers, function library, controller setup utilities, communication utilities, and program samples, place the PMC Motion CD into the PC computer CD drive. If the Motion CD does not auto start, browse the CD and select the file STARTUP.EXE.

Due to Windows Plug and Play issues, the MCAPI should not be installed 'on top of' previous installations. Please refer to **Removing the Motion Control API later** in this chapter.

The following windows should be displayed:

Step #1 - Select "Software and Manuals"

Step #2 - Select "PCI Bus Controllers"

Step 3) Select "DCX-PCI100 Controller"

Step #4) Choose Motion Control API

Step 5) Install Motion Control API

Step #6) Follow the on screen instructions

Motion Control API Components
Upon successful installation of PMC's Motion Control API, the Motion Control Panel will be available from the Windows Control Panel and the following components will be available from the Windows **Start** menu (Start\Programs\Motion Control\Motion Control API). For additional information on individual MCAPI components please refer to the **Software** and **Utilities** section in the **Programming, Software, and Utilities** chapter of this manual.

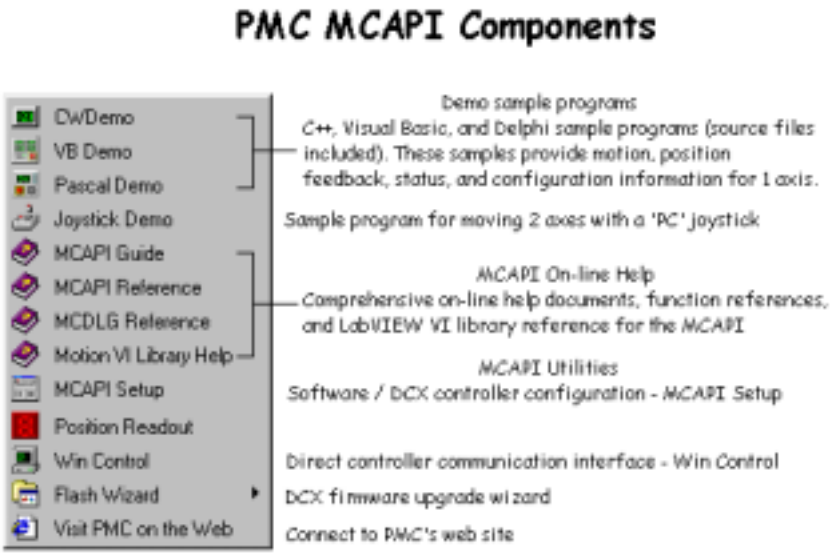


Figure 5: MCAPI components

Removing the Motion Control API
To remove the MCAPI , launch the Add/Remove Programs applet in the Windows Control Panel. After the Uninstall Shield has removed the MCAPI you will need to restart the computer to remove active .dll's.

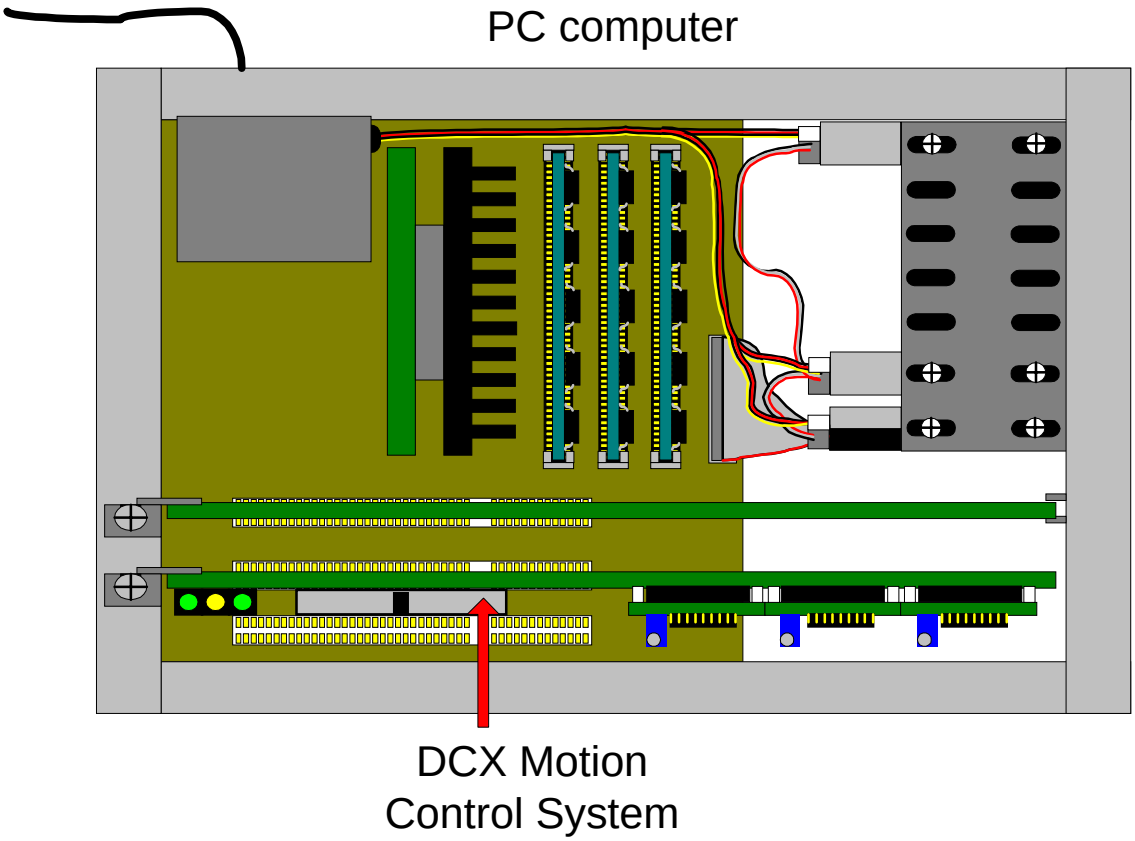
Figure 6: Windows Add/Remove programs

Installing the DCX-PCI100 Motion Control Motherboard

The DCX-PCI100 is 'Plug and Play' (Windows 2000/98/Me) compatible, there are no jumpers or switches to be configured. The DCX can be installed in any of the PC's available PCI slots. The DCX modules and cabling may interfere with a card installed in the slot next to the DCX, so it is recommended that the slot next to the DCX be left open. Make sure to attach the bracket of the DCX to the back panel of the PC.

Make sure that the PC computer power is turned off before installing the DCX-PCI100 motion controller.

For new installations, to verify communication between the PC, MCAPI, and the DCX it is recommended that the DCX-PCI100 motherboard first be installed **without any DCX modules**.



Plug & Play (Windows XP/2000/Me/98) Installation

The following section describes the basic steps for installing the DCX-PCI100 motion controller into plug and play PC computers. For step by step installation procedures please refer to the MCAPI read me file :\MotionCD\Windows\MCAPI\Current\Readme.txt

After installing the Windows driver (MCAPI 3.4.1 or higher), the DCX-PCI100 motion controller, and turning on the PC power the 'plug & play' operating system will detect a new PCI device.

Windows XP - The **Found New Hardware Wizard** will be launched (indicating that a new PCI device was detected). Proceed with the installation process by selecting:

Install the software automatically

If a windows list box of motion controllers / device drivers is displayed select the **PMC DCX-PCI100 Motion Controller**.

Note: Due to the considerable cost and maintenance overhead of Microsoft device driver qualification the PMC motion controller device drivers are not digitally signed.

Windows 2000 - Upon detecting a new PCI device Windows 2000 will automatically select the appropriate ***DCX-PCI100*** device driver. If the **Found New Hardware Wizard** is launched then the 'plug & play' installation has failed and you should contact PMC technical support.

Windows 98 - Upon detecting a new PCI device Windows 98 will automatically select the appropriate ***DCX-PCI100*** device driver. Note - Windows 98 does not handle 'plug & play installations as cleanly as XP & 2000. During the loading of the operating system a dialog may be displayed indicating the path to the MFX-PCI 1000 Series controllers device driver. Selecting **OK** will allow the 'plug & play' installation to be completed.

When the operating system has completed loading, launch the **Windows Device Manager**. Select **Hardware** and then **Motion Control**. The Device Manager should list the ***DCX-PCI100 Motion Controller*** as an installed device.

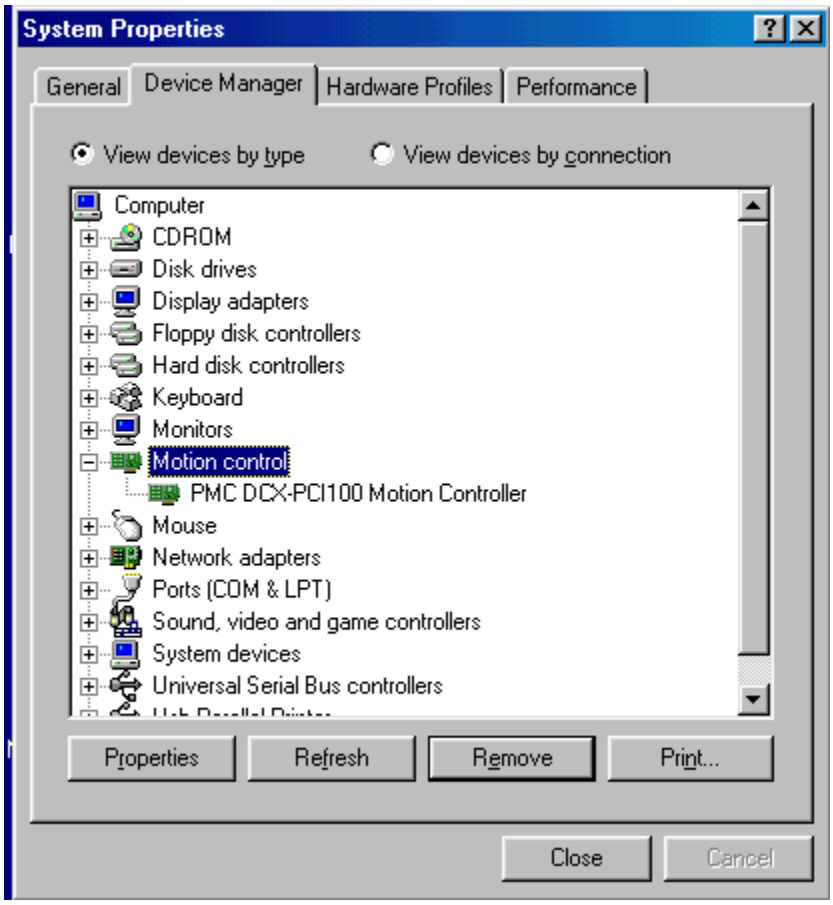


Figure 7: Use the Windows Device Manger to verify 'plug & play' installation

Verify Communication with the PC

The final step of a DCX-PCI100 instillation is to verify communication between the PC and the motion controller. This can be accomplished via either:

The Motion Control Panel applet
Win Control Terminal Emulator

Motion Control Panel applet
From the Motion Control panel (Start\Settings\Control Panel\Motion Control) you can view the installed versions of the Motion Control API and the on-board firmware of the DCX-PCI100 controller. To report the software and firmware versions select **Properties** and then **Info**. The MCAPI will query the DCX controller for its firmware version. If the Motion Control Panel is unable to acquire this information the version will be reported as unknown.

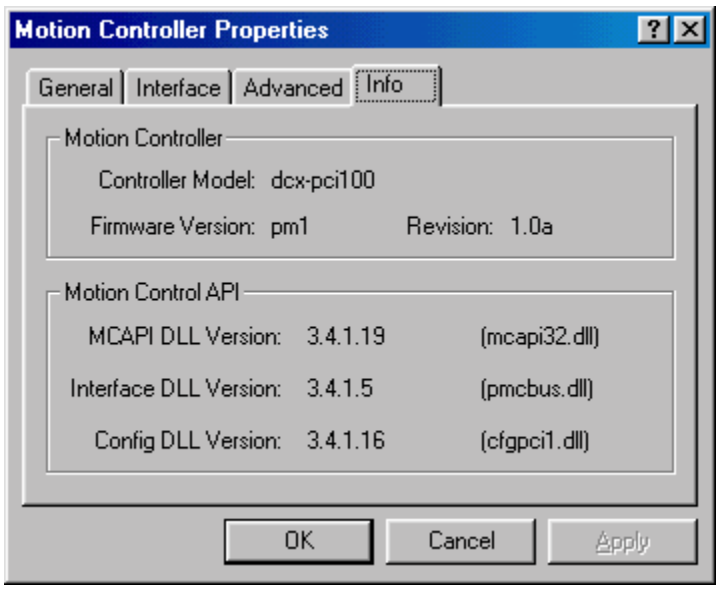


Figure 8: Checking firmware and MCAPI version

Win Control Terminal Emulator
From the Windows Start Menu select:

\Programs\Motion Control\Motion Control API\Win Control

If WinControl program opens and reports the firmware version of the ***MFX-PCI*** the controller and MCAPI software have been properly installed and basic communication has been verified.

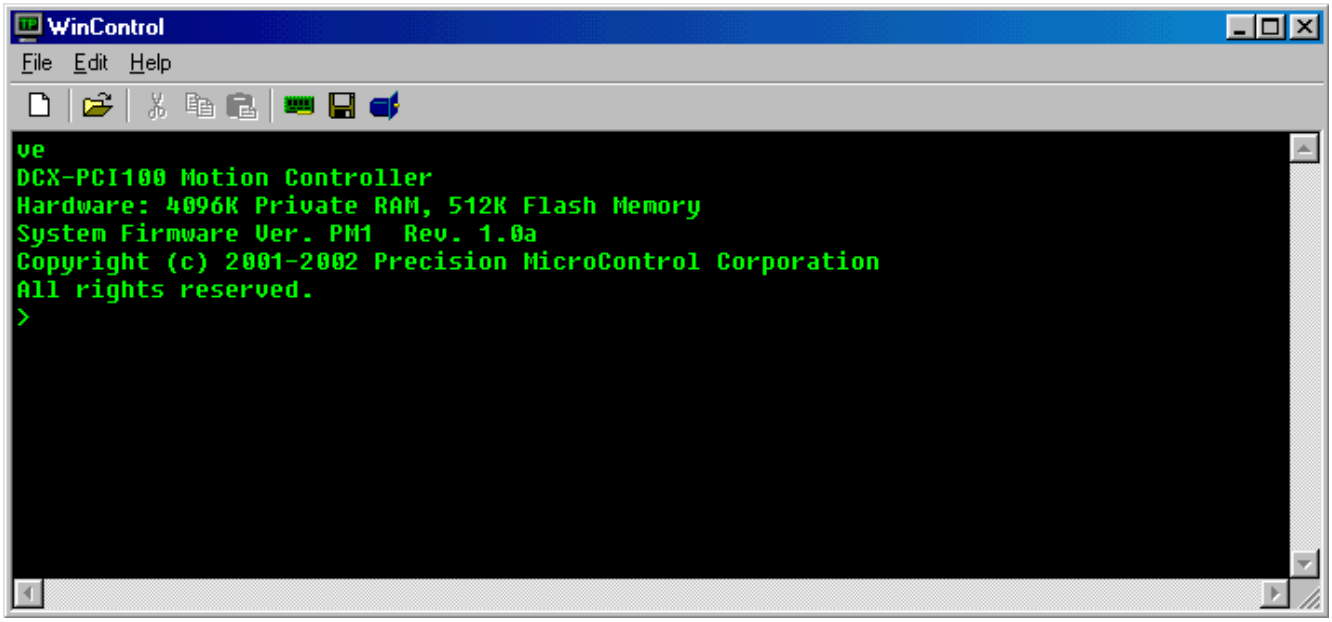


Figure 9: Use WinControl to verify controller communication

If an error message is displayed the PC / MCAPI / ***DCX-PCI100*** are not communicating properly and an error message will be returned and you should contact PMC Technical Support..

Figure 10: Failed communication error message

Windows NT Installation

There are no jumpers or switches to be configured prior to installing the DCX-PCI100 in a Windows NT PC. The DCX can be installed in any of the PC's available PCI slots. The DCX modules and cabling may interfere with a card installed in the slot next to the DCX, so it is recommended that the slot next to the DCX be left open . Make sure to attach the bracket of the DCX to the back panel of the PC.

Make sure that the PC computer power is turned off before installing the DCX-PCI100 motion controller.

For new installations, to verify communication between the PC, MCAPI, and the DCX it is recommended that the DCX-PCI100 motherboard first be installed without any DCX modules. After installing the DCX-PCI100, turn on the PC and log on to the Windows NT system as the **system administrator**.

To install PMC's motion control software, the MCAPI, the user must be logged on as the system administrator.

For assistance with installing the MCAPI please refer to the section titled **Installing the DCX Software (MCAPI) on page 16**.

Windows NT is **not a 'plug & play' operating system**. the user must configure the MCAPI device driver for the type and quantity of DCX-PCI100 controllers installed in the computer. The next few pages describe the steps required to configure the MCAPI.

Launch PMC's **New Controller Wizard** by selecting the Motion Control icon from the Windows Control Panel or from the Windows **Start** menu (Motion Control\Motion Control API\MCAPI Setup).

Figure 11:For NT systems launch Motion Control from the Windows Control Panel

Do not attempt to setup the Motion Control API without a DCX-PCI100 motion controller installed in the PC. The last step of the **New Controller Wizard** verifies communication between the DCX controller and the PC.

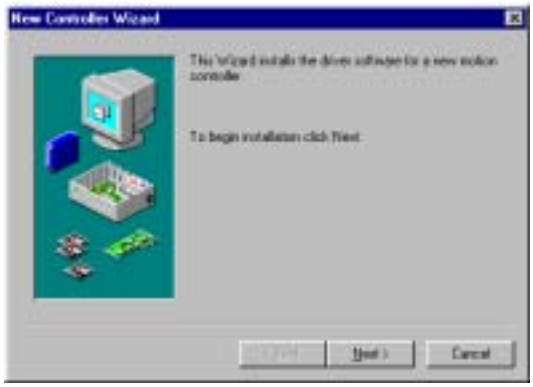


Figure 12:PMC's New Controller Wizard

Controller ID

Each PMC motion controller installed in your PC requires an individual Controller ID number. The MCAPI supports controller ID's between 0 and 15, supporting applications with as many as 16 DCX controllers in a single computer. Typically the Controller ID is set to zero (ID=0). If more than one DCX controller is to be installed usually the DCX-PCI100 upon which the primary axes reside is set to ID0.

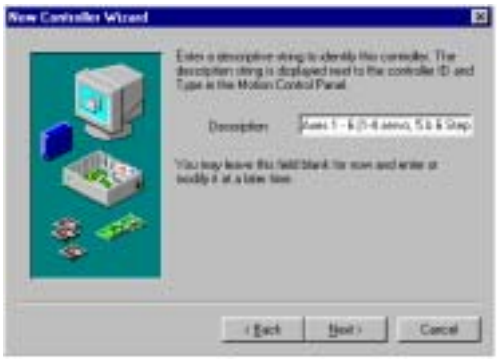
Figure 13: Setting the Controller ID

Controller Type

The MCAPI supports mixing and matching various PMC controllers (DCX-PCI100, DCX-PC100, and DC2-PC) within a single PC. A list of PMC controllers that are supported by the MCAPI will be displayed. Select the DCX-PCI100.

Figure 14: Select Controller Type - DCX-PCI100

Description
Allows the user to enter comments about the controller. An example of a completed General setup of a DCX-PCI100 follows:



Communications Interface
A list of supported controller interfaces will be displayed. Select the PC-Bus.

Figure 15: Selecting the communication Interface

Testing the Installation
To verify the DCX / MCAPI installation open the WinControl32 utility (Start\Programs\Motion Control\Motion Control API\Win Control). If WinControl opens and reports the firmware version of the

DCX the system is operating properly. If the PC / MCAPI / DCX are not communicating properly an error message will be returned.

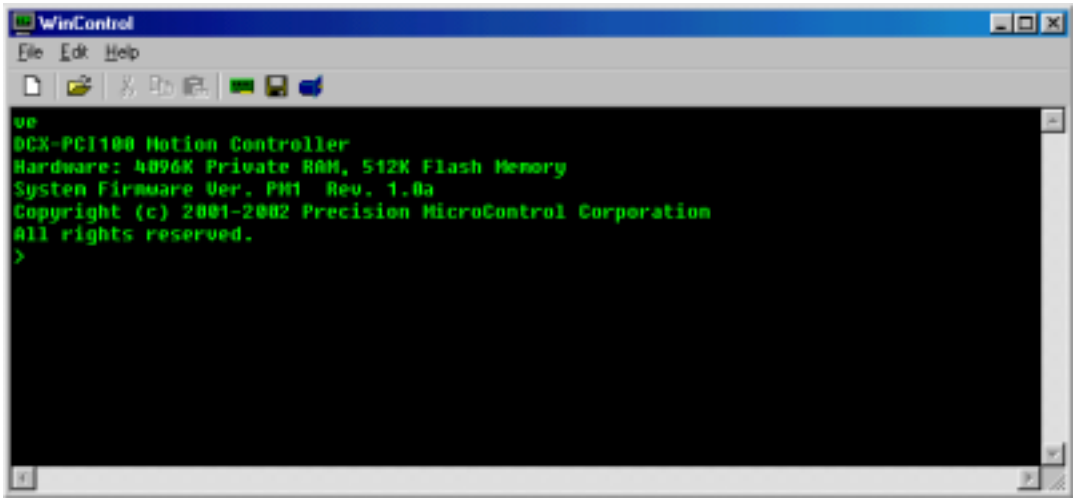


Figure 16: Use WinControl to verify controller / MCAPI / computer communication

Figure 17: Controller communication failed error message

If WinControl fails contact PMC Technical Support.

Chapter Contents

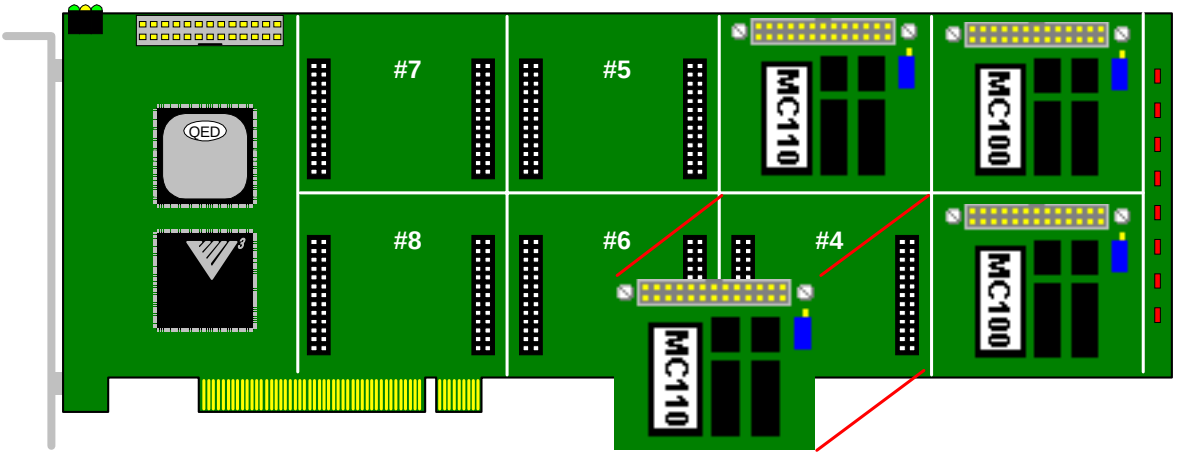
Chapter 3

DCX Module Installation

Installing DCX Motor Control and I/O Modules

DCX Modules can be placed in any open module position on the DCX motherboard. If there are fewer than eight modules to be installed on the DCX, spread them out as much as possible. This will allow easier installation and removal of the modules as well as mating cables.

If there are to be motor control modules installed on the DCX, and you want them to be numbered in a specific order, install them in module positions on the DCX in that order. For example, the module that is to control motor number 1 could be installed in module position number 1 (refer to the module numbers on the DCX circuit board). The module controlling motor number 2 could be installed in position number 2, and so on. Alternatively, the second module could be installed in any other module position and it will still be assigned number 2 since it is the second motor module on the DCX.

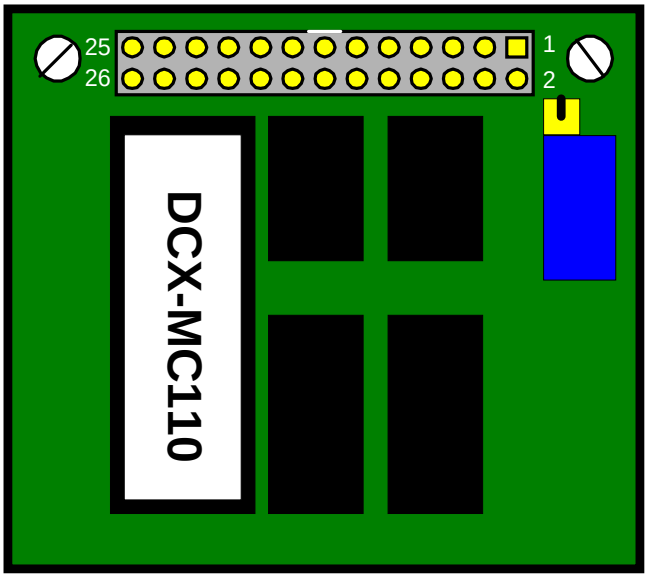


To install the modules, lay the DCX-PCI100 motherboard on a flat surface, component side up. Place each DCX module in the desired position, aligning the connectors and mounting holes with their respective mates on the DCX motherboard. When you are satisfied that the module is properly aligned, carefully press the module into the DCX. The header pins of the module should seat completely into the mating connectors on the DCX motherboard. Two nylon mounting screws are supplied with each DCX module. These should be installed from the backside of the motherboard, into the standoffs on the modules. Repeat this process for installing modules on the DCX until all modules are in place.

Next the DCX should be re-installed in the PC chassis and interfacing cables connected. Refer to the following sections in this chapter for specific jumper and wiring information for the types of modules that are being used. When cabling has been completed, power can be applied to the system and initial checkout can begin.

Please note that all DCX modules contain a 26 pin, shrouded, center polarized header for I/O connections. The pins of this connector are numbered from 1 to 26. The following diagram shows the location of pins 1, 2, 25 and 26. The other 22 pins are numbered and located respectively.

DCX MODULE CONNECTOR PIN NUMBERING (TOP SIDE VIEW)



DCX-MC100 – Servo Motor Module Installation

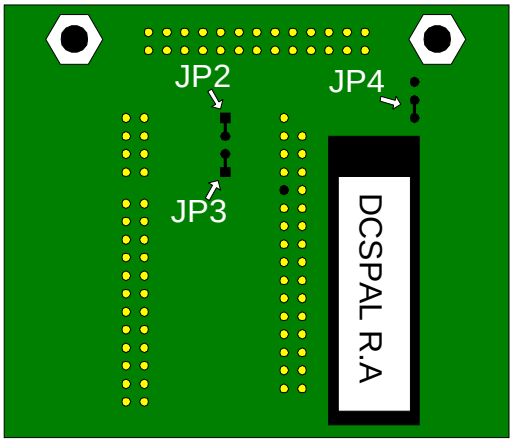
The default shipping configuration for the DCX-MC100 supports:

- +/- 10 volt servo command output (12 bit resolution, 10 ma. Max.)
- Single ended encoder (phase A, phase B)
- Encoder Index Z- (TTL level, low active)
- Coarse Home, Limit + , Limit – , and Amplifier Fault inputs (TTL level, low active)
- Amplifier Inhibit output (TTL level, low active, 10 ma max.)
- +5 VDC encoder power output (100 ma max.)
- Servo Command output offset adjustment potentiometer

+12 volt motor drive power supply
The default configuration of the DCX-PCI100 does not use the +12 volt connection on the PCI bus edge connector. To supply +12 volts to the DCX-PCI100 the user must connect J33 to a PC computer disk drive power supply connector. Typically a standard disk drive power supply 'splitter' cable is used to connect the +12 volt supply of the PC computer to the DCX controller. Power supply splitter cables can be purchased from PMC (P/N 71.060.A). For additional information please contact the factory.

If the +12 volt PC computer power supply connection is not provided to the DCX-PCI100 J33 connector **no servo motion will occur**.

Differential Encoder
The DCX-MC100 can be configured to support a differential encoder by cutting the signal traces between pins 1 and 2 of JP2 and JP3 (back side of module).



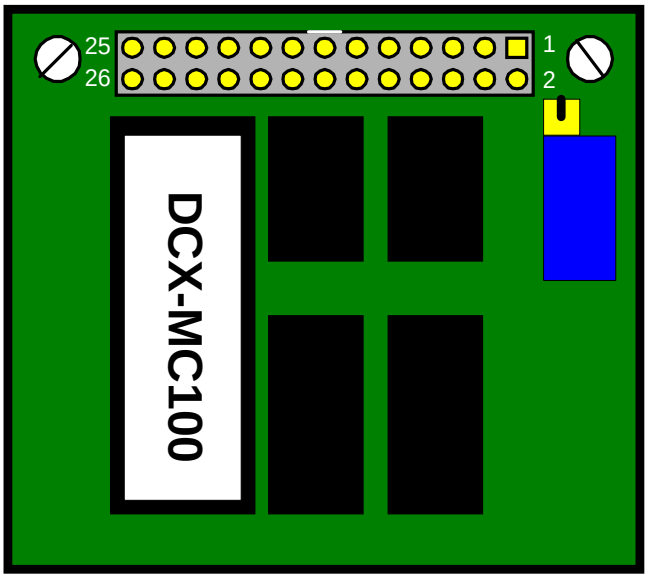
+12 volt encoder power
The DCX-MC100 can be configured to provide a +12 VDC Encoder Power Output by:

1) Cutting the signal trace between pins 2 and 3 of JP4 and
2) Connecting pins 1 and 2 of JP4

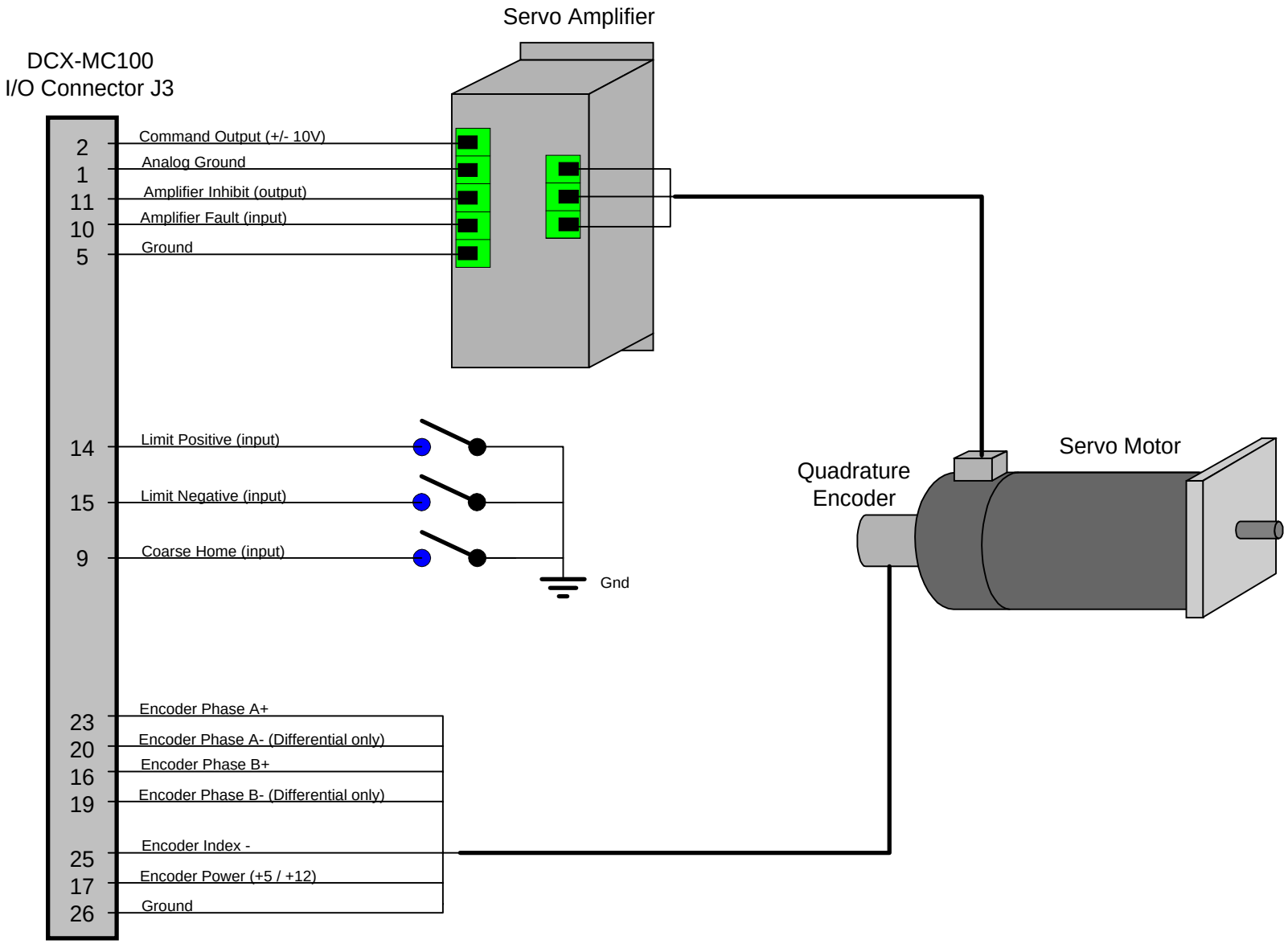
Note: The DCX-MC100 provides the Encoder Power output as a convenience, It is **not required** that it be used to power the encoder.

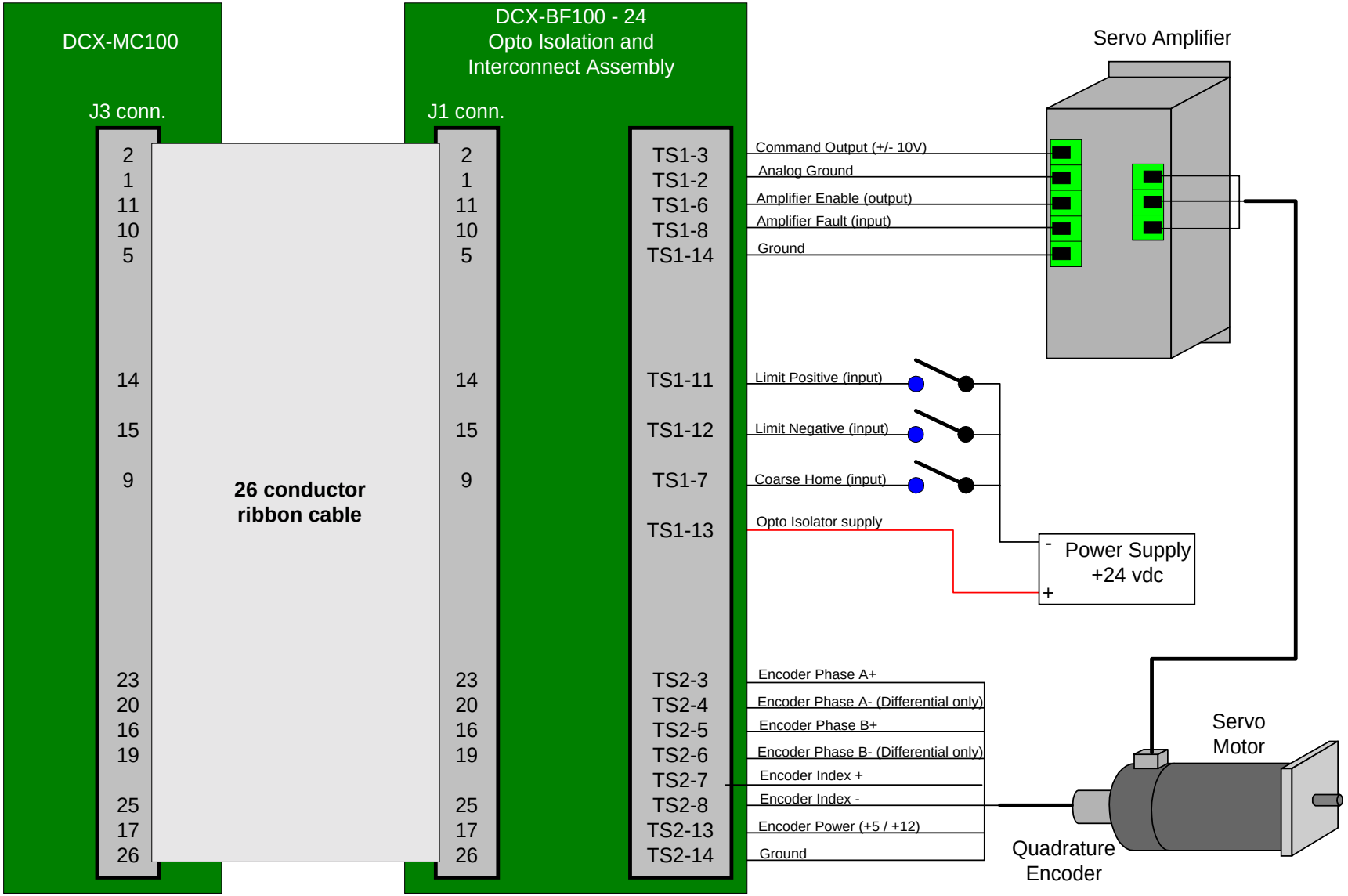
All external connections (Command signal, Limits, encoder, etc…) are made via the 26 pin, dual row header labeled J1. The diagram below details the pin number of the J3 connector.

DCX-MC100 J3 CONNECTOR PIN NUMBERING (TOP SIDE VIEW)



After installing the DCX-MC100 module into the DCX-PCI100 motion control motherboard the servo encoder, amplifier, and limit switches can be connected to the module. Wiring diagrams on the next two pages depict typical installations. The first diagram details direct connection of the MC100 to the external components (servo amplifier, encoder, and sensors). The second diagram details typical connections when a **DCX-BF100 Opto Isolation and Interconnect Assembly** is used.


Servo Amplifier
DCX-MC100
I/O Connector J3
2
1
11
10
5
Command Output (+/- 10V)
Analog Ground
Amplifier Inhibit (output)
Amplifier Fault (input)
Ground
14
15
9
Limit Positive (input)
Limit Negative (input)
Coarse Home (input)
Gnd
Servo Motor
Quadrature
Encoder
23
20
16
19
25
17
26
Encoder Phase A+
Encoder Phase A- (Differential only)
Encoder Phase B+
Encoder Phase B- (Differential only)
Encoder Index -
Encoder Power (+5 / +12)
Ground


DCX-MC100
J3 conn.
2
1
11
10
5
14
15
9
23
20
16
19
25
17
26
26 conductor ribbon cable
DCX-BF100 - 24
Opto Isolation and Interconnect Assembly
J1 conn.
2
1
11
10
5
14
15
9
23
20
16
19
25
17
26
TS1-3
TS1-2
TS1-6
TS1-8
TS1-14
TS1-11
TS1-12
TS1-7
TS1-13
TS2-3
TS2-4
TS2-5
TS2-6
TS2-7
TS2-8
TS2-13
TS2-14
Servo Amplifier
Command Output (+/- 10V)
Analog Ground
Amplifier Enable (output)
Amplifier Fault (input)
Ground
Limit Positive (input)
Limit Negative (input)
Coarse Home (input)
Opto Isolator supply
- Power Supply +24 vdc +
Encoder Phase A+
Encoder Phase A- (Differential only)
Encoder Phase B+
Encoder Phase B- (Differential only)
Encoder Index +
Encoder Index -
Encoder Power (+5 / +12)
Ground
Servo Motor
Quadrature Encoder

DCX-MC110 – Servo Motor Module Installation

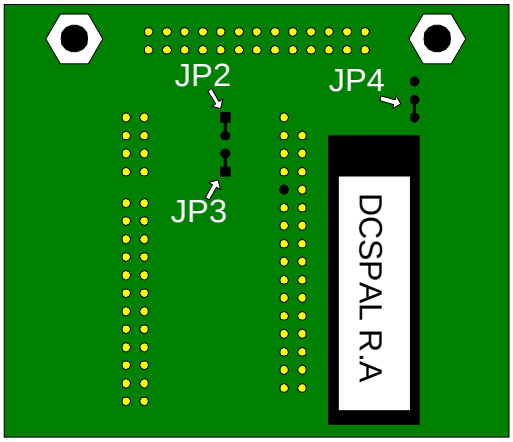
The default shipping configuration for the DCX-MC110 supports:

- 0 - 12 volt motor drive output (8 bit resolution, 500 ma. Max.)
- Single ended encoder (phase A, phase B)
- Encoder Index Z- (TTL level, low active)
- Coarse Home, Limit + , Limit – , and Amplifier Fault inputs (TTL level, low active)
- Amplifier Inhibit output (TTL level, low active, 10 ma max.)
- +5 VDC encoder power output (100 ma max.)
- Motor Drive output offset adjustment potentiometer

+12 volt motor drive power supply
The PCI bus motherboard edge connector was not designed to provide high current to accessory cards like the DCX-PCI100. In order to provide sufficient supply voltage / current for DCX-MC110 motor drive modules (0.5 amps per module, maximum of 4.0 amps) a 4 pin connector (J33) matching the power supply pinout of 5 ¼ / Hard Disk Drives can be found on the DCX-PCI100 motherboard. A standard disk drive power supply 'splitter' cable is used to connect the +12 volt supply of the PC computer to the DCX controller. Power supply splitter cables can be purchased from PMC (P/N 71.060.A)

If the +12 volt PC computer power supply connection is not provided to the DCX-PCI100 J33 connector **no servo motion will occur**.

Differential Encoder
The DCX-MC110 can be configured to support a differential encoder by cutting the signal traces between pins 1 and 2 of JP2 and JP3 (back side of module).



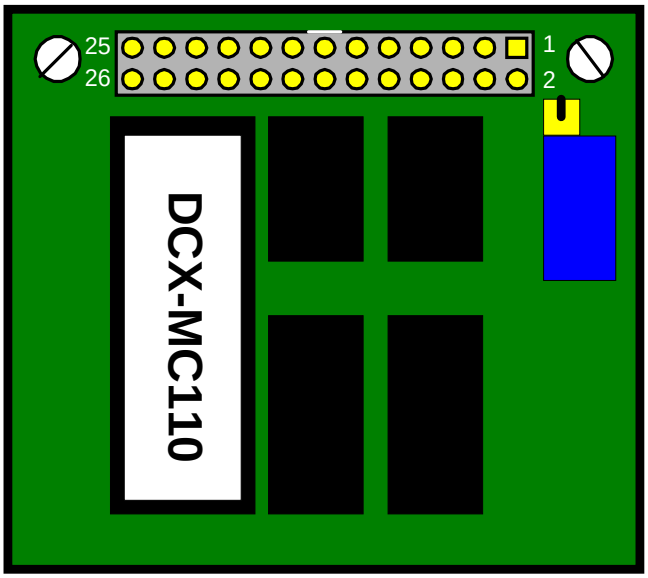
+12 volt encoder power
The DCX-MC110 can be configured to provide a +12 VDC Encoder Power Output by:

3) Cutting the signal trace between pins 2 and 3 of JP4 and
4) Connecting pins 1 and 2 of JP4

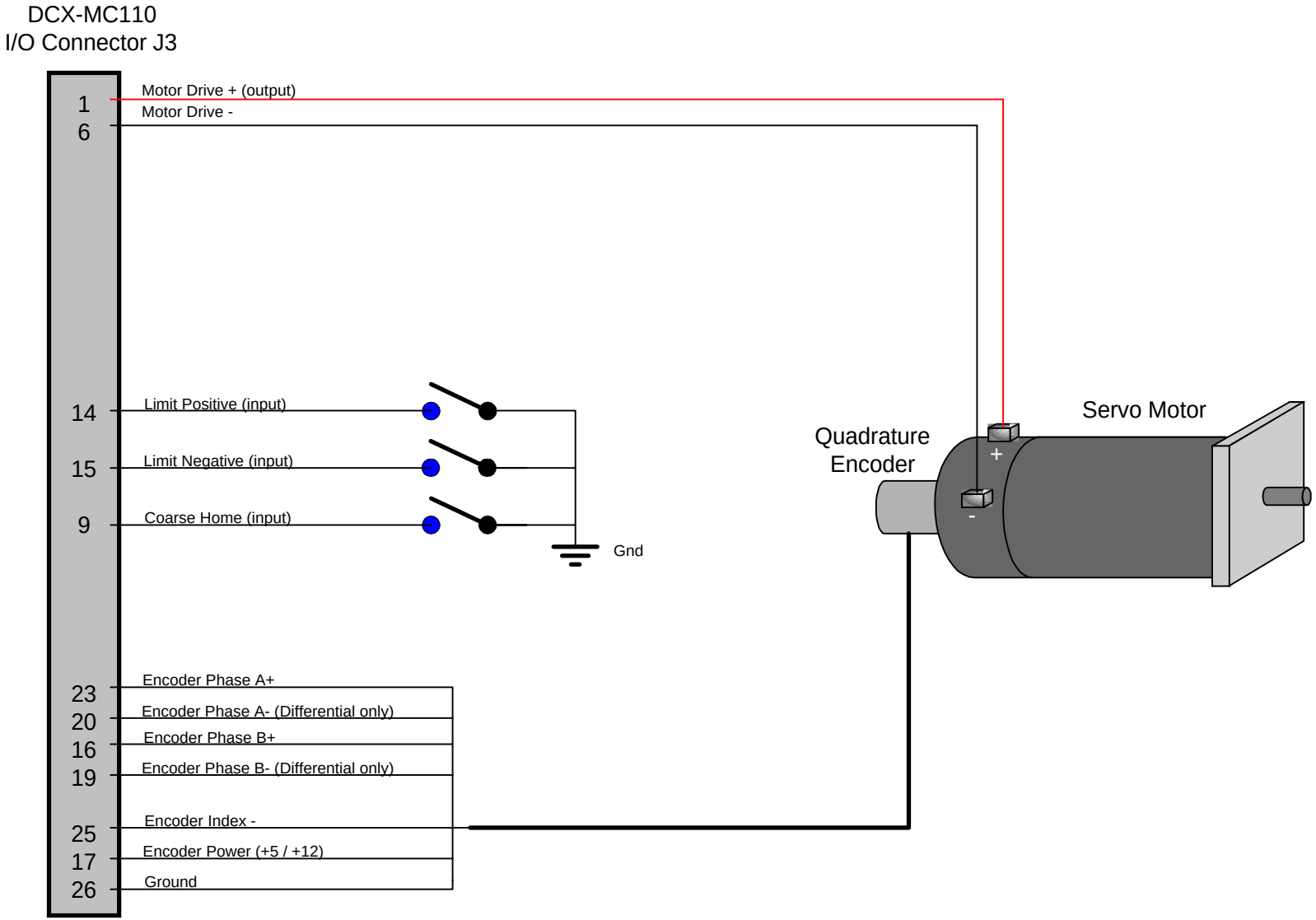
Note: The DCX-MC110 provides the Encoder Power output as a convenience, It is **not required** that it be used to power the encoder.

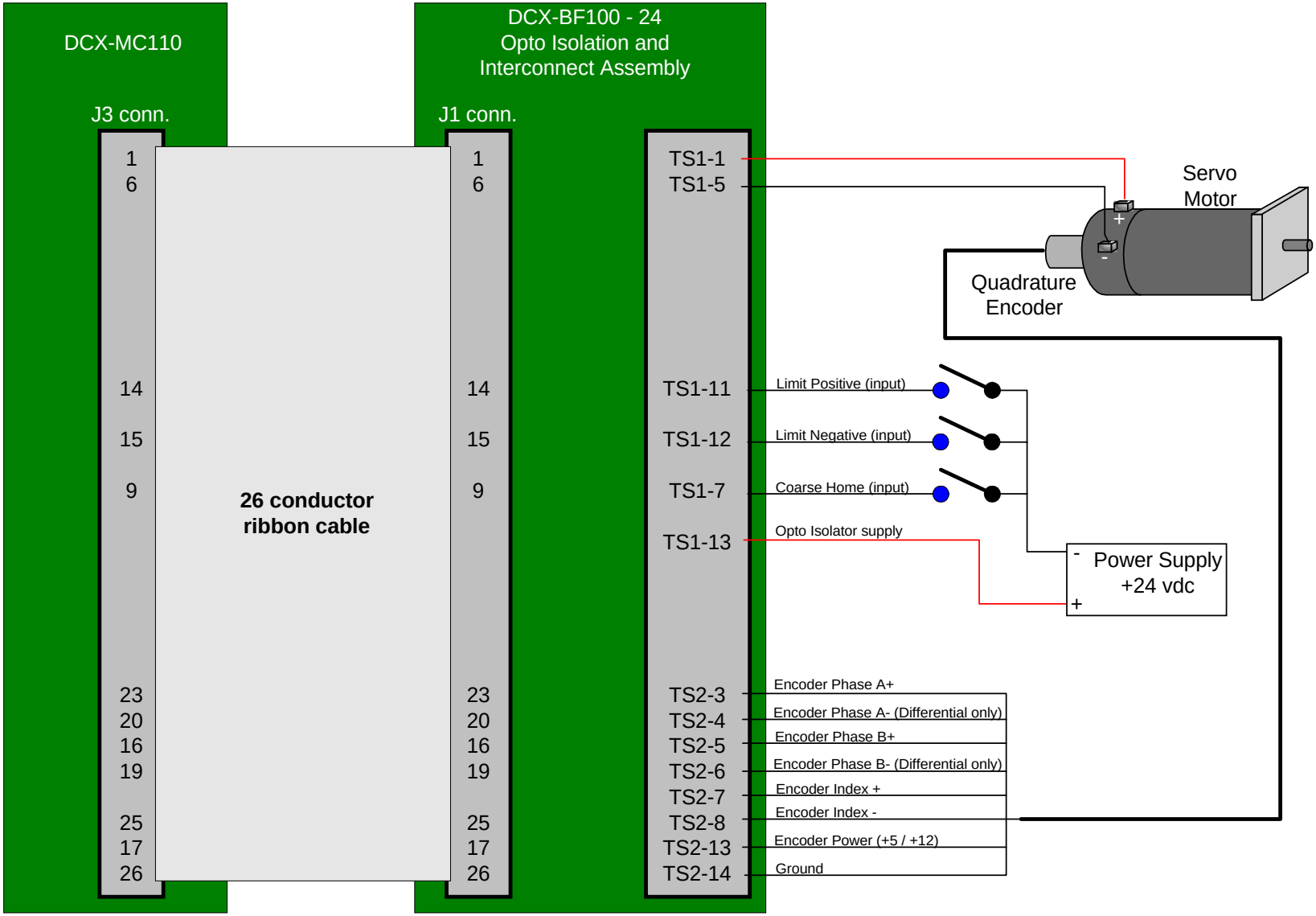
All external connections (Command signal, Limits, encoder, etc…) are made via the 26 pin, dual row header labeled J1. The diagram below details the pin number of the J3 connector.

DCX-MC110 J3 CONNECTOR PIN NUMBERING (TOP SIDE VIEW)



After installing the DCX-MC110 module into the DCX-PCI100 motion control motherboard the encoder, motor, and limit switches can be connected to the module. Wiring diagrams on the next two pages depict typical installations. The first diagram details direct connection of the MC110 to the external components (motor amplifier, encoder, and sensors). The second diagram details typical connections when a **DCX-BF100 Opto Isolation and Interconnect Assembly** is used.

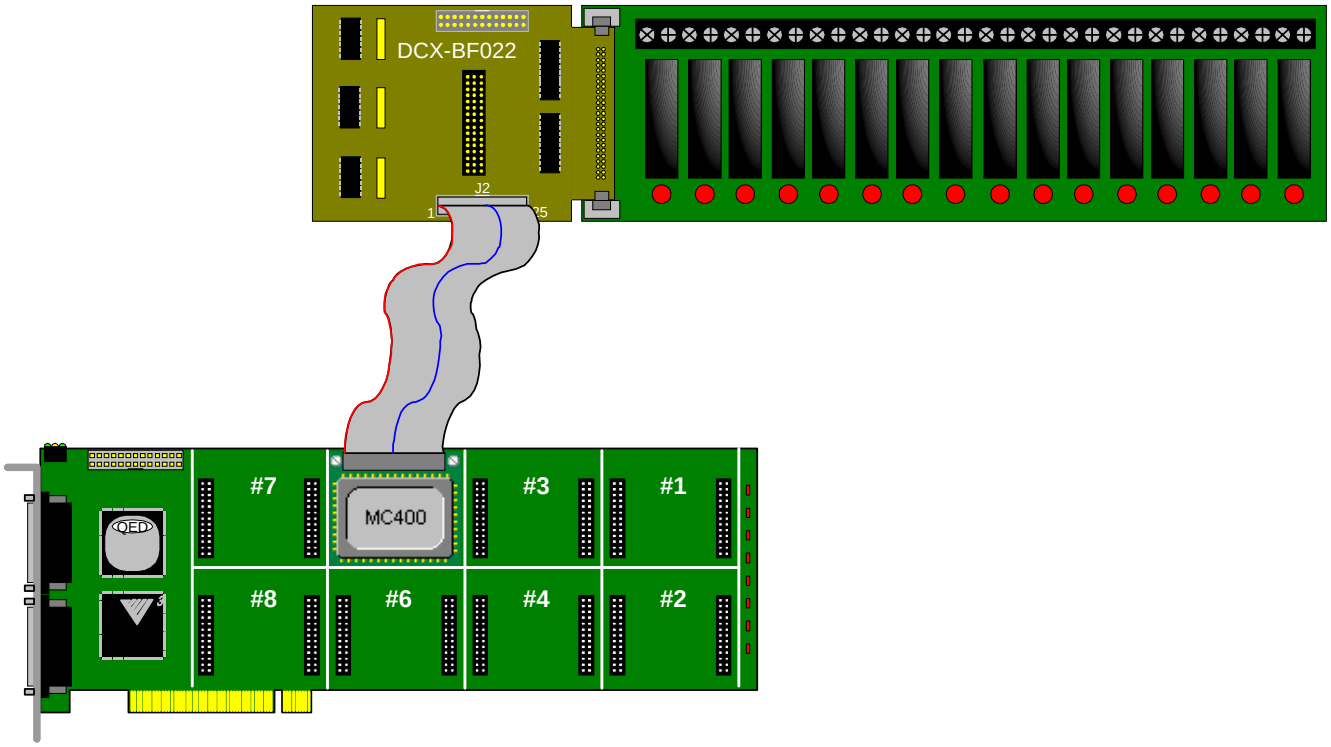

DCX-MC110
I/O Connector J3
1
6
Motor Drive + (output)
Motor Drive -
14
Limit Positive (input)
15
Limit Negative (input)
9
Coarse Home (input)
Gnd
Quadrature
Encoder
Servo Motor
+
-
23
Encoder Phase A+
20
Encoder Phase A- (Differential only)
16
Encoder Phase B+
19
Encoder Phase B- (Differential only)
25
Encoder Index -
17
Encoder Power (+5 / +12)
26
Ground


DCX-MC110
J3 conn.
1
6
14
15
9
23
20
16
19
25
17
26
26 conductor ribbon cable
DCX-BF100 - 24
Opto Isolation and Interconnect Assembly
J1 conn.
1
6
14
15
9
23
20
16
19
25
17
26
TS1-1
TS1-5
TS1-11
TS1-12
TS1-7
TS1-13
TS2-3
TS2-4
TS2-5
TS2-6
TS2-7
TS2-8
TS2-13
TS2-14
Servo Motor
+
-
Quadrature Encoder
Limit Positive (input)
Limit Negative (input)
Coarse Home (input)
Opto Isolator supply
- Power Supply +24 vdc +
Encoder Phase A+
Encoder Phase A- (Differential only)
Encoder Phase B+
Encoder Phase B- (Differential only)
Encoder Index +
Encoder Index -
Encoder Power (+5 / +12)
Ground

DCX-MC400 – Digital I/O Expansion Module Installation

One or more MC400 digital I/O modules can be installed on the DCX. There are no jumpers on this module to be configured. The module's TTL digital I/O signals can be connected directly to the external circuits if output loading (1ma maximum sink/source)and input voltages are within acceptable limits. Alternatively, a BFO22 interface board can be used to connect the module's I/O to a relay rack in order to provide optically isolated inputs and outputs.

The BFO22 interface board provides a convenient means of connecting the MC400's TTL digital I/O channels to a 16 position relay rack available from two manufacturers, Opto22 (P/N PB16H) and Grayhill (P/N 70RCK16-HL). These relay racks accept up to 16 optically isolated input or output modules for interfacing with external electrical systems. Using one of these relay racks and a BFO22, an optically isolated I/O module can be connected to each of the MC400's digital I/O channels.



As shown above, the BFO22 plugs directly into the relay rack's 50 pin header connector and then connects to the MC400 via a 26 conductor ribbon cable. Note that the relays are numbered sequentially starting from 0, while the DCX digital I/O channels are numbered sequentially starting with 1.

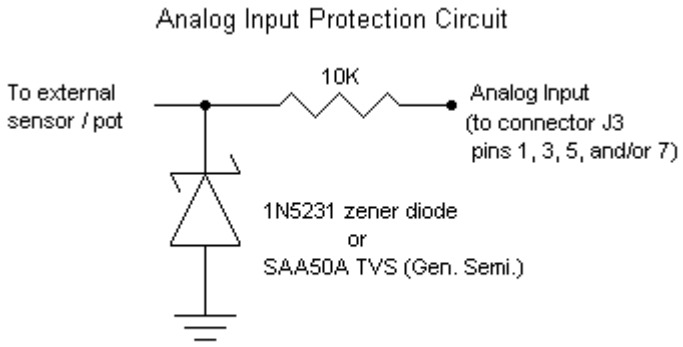
Although the relay rack has screw terminals for connecting a logic supply, it is not necessary to make this connection. By installing a shorting block on jumper JP17 of the BFO22, the 5 volt supply of the DCX will be supplied to the relay rack.

For detailed information on configuring the DCX-BF022, please refer to the schematic and jumper table in the DCX-BF022 Appendix in this user manual.

DCX-MC500 – Analog I/O Expansion Module Installation

One or more MC500 analog I/O modules can be installed in the DCX as described in the first section of this chapter. There are no jumpers on this module to be configured. The module's I/O signals can be connected directly to the user's external circuits as long as output loading is not excessive and input voltages are maintained within the specified limits (see the MC500 appendix).

A voltage level greater than 5.6 volts will damage DCX-MC500 analog input channels. The schematic below is recommended to protect an analog input from damage due to an over voltage condition. This circuit will limit the maximum voltage applied to the A/D converter to 5.6 VDC.



Chapter Contents

- Introduction to the Motion Control Application Programming Interface (MCAPI)
- Controller Interface Types
- Building Application Programs using MCAPI
  - C++ programming
  - Visual Basic Programming
  - Delphi Programming
  - LabVIEW programming
- PMC Sample Programs
- Motion Integrator
  - System Integration Wizards
  - Servo Tuning tool
  - Embeddable OLE servers
- PMC Utilities
  - MCAPI Setup
  - WinControl
  - FlashWizard
  - Joystick Applet
  - Position Readout
- MCAPI On-line Help
  - MCAPI Users Guide
  - MCAPI on-line function reference
  - MCAPI Common Dialog help
  - LabVIEW Motion VI Library Help

Chapter 4

Programming, Software and Utilities

The DCX motion control system integrates seamlessly into high performance, Windows applications. The **Motion Control Application Programming Interface** (**MCAPI**) provides support for all popular high level languages. Additionally, the board level command set (MCCL) allows the machine designer to execute local 'macro' routines independent of the PC host and its application programs.

PMC's Motion Control API (MCAPI) is a group of Windows components that, taken together, provide a consistent, high level, Applications Programming Interface (API) for PMC's motion controllers. The difficulties of interfacing to new controllers, as well as resolving controller specific details, are handled by the API, leaving the applications programmer free to concentrate on the application program.

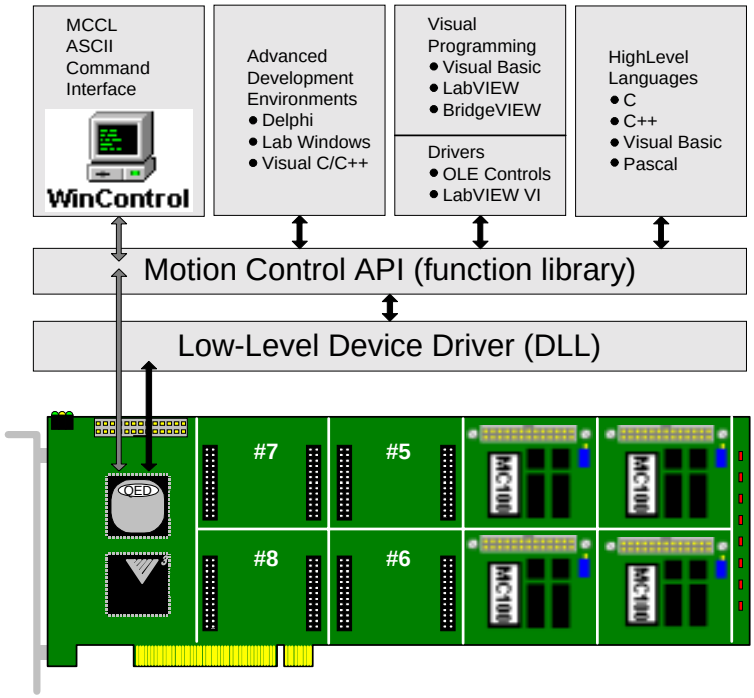


Figure 18: MCAPI and DCX-PCI100 architectural diagram

The API has been constructed with a layered approach. As new versions of Windows operating systems and new PMC motion controllers become available API support is provided by simply replacing one or more of these layers. Because the public API (the part the applications programmer sees) is above these layers, few or no changes to applications programs will be required to support new version of the MCAPI.

The API itself is implemented in three parts. The low level device driver provides communications with the motion controller, in a way that is compatible with the Microsoft Windows operating system. The MCAPI low level driver passes binary MCCL commands (Motion Control Command Language – the instruction set of the DCX motion controller) to the DCX. By placing the operating system specific portions of the API here it will be possible to replace this component in the future to support new operating systems without breaking application programs, which rely on the upper layers of the API.

Sitting above that, and communicating with the driver is the API Dynamic Link Library (DLL). The DLL layer implements the high level motion functions that make up the API. This layer also handles the differences in operation of the various PMC Motion Controllers, making these differences virtually transparent to users of the API.

At the highest level are environment specific drivers and support files. These components support specific features of that particular environment or development system.

Care has been exercised in the construction of the API to ensure it meets with Windows interface guidelines. Consistency with the Windows guidelines makes the API accessible to any application that can use standard Windows components - even those that were developed after the Motion Control API. A Quick Reference Guide and detailed MCAPI Function Library Listing can be found in the manual.

Controller Interface Types

The DCX controller supports two onboard interfaces, an ASCII (text) based interface and a binary interface. The binary interface is used for high speed command operation, and the ASCII interface is used for interactive text based operation (WinControl). The high level sample programs (CWDEMO and VBDEMO) use the binary interface, PMC WinControl uses the ASCII interface.

Application programs must indicate which interface they intend to use when they open a handle for a particular controller. A controller may have more than one handle open at a time, but all open handles for a particular controller must specify the same interface (all must be open with the binary interface or all must be open with the ASCII interface). The open mode is specified by setting the second argument of the ***MCOpen( )*** function to either **MC_OPEN_ASCII** or **MC_OPEN_BINARY**.

Note that not all functions are available in the ASCII mode of operation, this mode is intended primarily for use with the ***pmcgetc( )***, ***pmcgets( )***, ***pmcputc( )***, and ***pmcputs( )*** character based functions (these 4 functions are not available in binary mode). This restriction will be eliminated in a future release of the API.

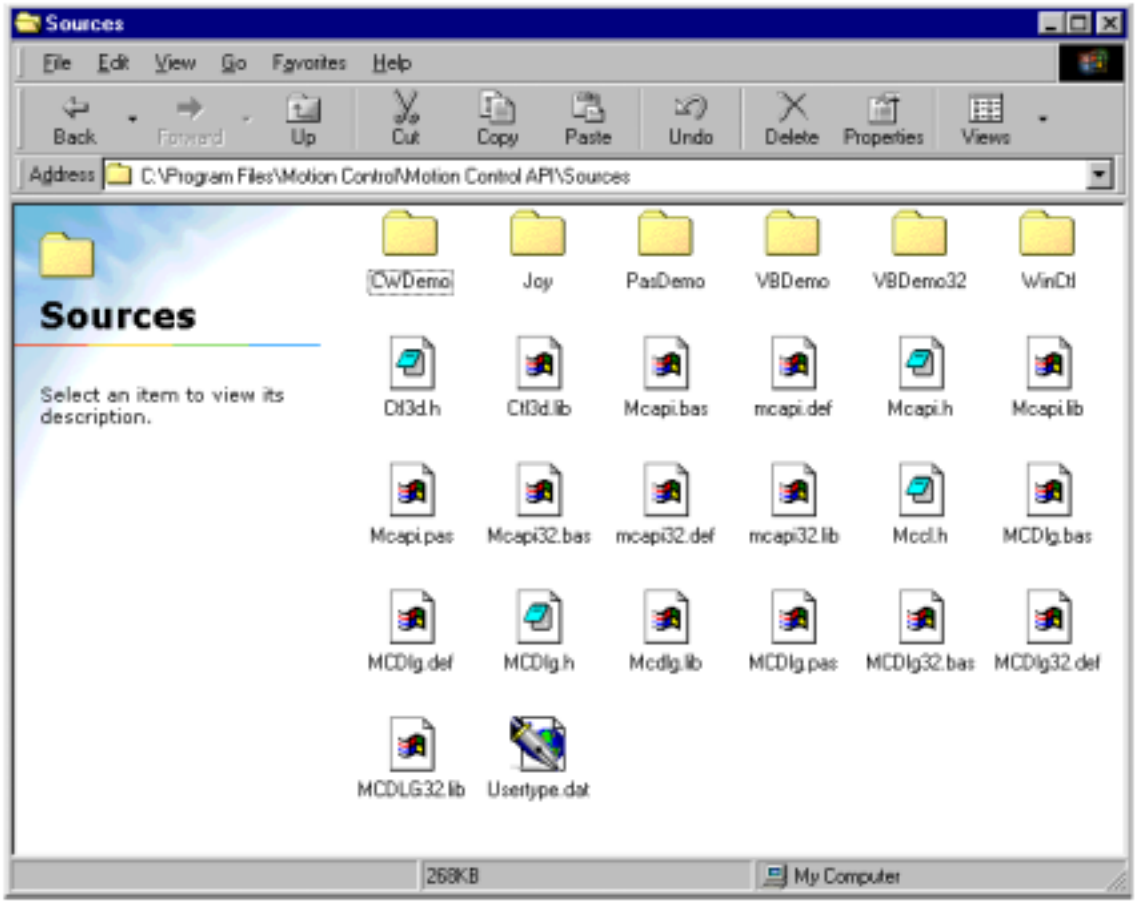
Building Application Programs using Motion Control API

The Motion Control Application Programming Interface (MCAPI) is designed to allow a programmer to quickly develop sophisticated application programs using popular development tools. The MCAPI provides high level function calls for:

- Configuring the controller (servo tuning parameters, velocity and ramping, motion limits, etc.)
- Defining on-board user scaling (encoder units, velocity units, dwell time units, user and part zero)
- Commanding motion (Point to Point, Constant velocity)
- Reporting controller data (motor status, position, following error, current settings)
- Monitoring Digital and Analog I/O
- Driver functions (open controller handle, close controller handle, set timeout)

A complete description of all MCAPI functions can be found in later in this manual.

Included with the installation of the MCAPI is the Sources ‘folder’. In this folder are complete program sample source files for C++, VisualBasic, and Delphi.



C/C++ Programming

Included with each of the C program samples (CWDemo. Joystick demo, and WinControl) is a read me file (readme.txt) that describes how to build the sample program. The following text was reprinted from the readme.txt file for the CWDemo program sample.

Contents
========
- How to build the sample
- LIB file issues
- Contacting technical support

How to build the sample
=======================
To build the samples you will need to create a new project or make file within your C/C++ development tool. Include the following files in your project:

CWDemo.c
CWDemo.def
CWDemo.rc

For 16-bit development you will also need:

..\mcapi.lib
..\mcdlg.lib
..\ctl3d.lib

For 32-bit development you will also need:

..\mcapi32.lib
..\mcdlg32.lib

If your compiler does not define the _WIN32 constant for 32-bit projects you will need to define it at the top of the source file (before the header files are included).

LIB File Issues
===============
Library (LIB) files are included with MCAPI for all the DLLs that comprise the user portion of the API (MCAPI.DLL, MCAPI32.DLL, MCDLG.DLL, and MCDLG32.DLL). These LIB files make it easy to resolve references to functions in the DLL using static linking (typical of C/C++). Unfortunately, under WIN32 the format of the LIB files varies from compiler vendor to compiler vendor. If you cannot use the included LIB files with your compiler you will need to add an IMPORTS section to your projects DEF file. We have included skeleton DEF files for all of the DLLs for which we also include a LIB file (MCAPI.DEF, MCAPI32.DEF, MCDLG.DEF, and MCDLG32.DEF).

The 16-bit LIB files were built with Microsoft Visual C/C++ Version 1.52, and the 32-bit LIB files Microsoft Visual Studio Version 5.

Visual Basic Programming

Included with each of the Visual Basic program samples (VBDemo. VBDemo32) is a read me file (readme.txt) that describes how to build the sample program. The following text was reprinted from the readme.txt file for the VBDemo32 program sample.

Contents
========

- About the sample
- How to build the sample
- Contacting technical support

About the sample
================
This sample demonstrates a simple user interface to one axis of a motion controller. The user may program moves and interact with the motion in a number of ways (stop it, abort it, etc.). Sample forms demonstrate how to configure servo or stepper motor axes. A number of the new MCDialog functions (such as a full-featured, ready-to-run
axis configuration dialog) are also demonstrated.

How to build the sample
=======================
To build the samples you will need to create a new project or use the Visual Basic project file (created with Visual Basic v6.0) included with the sample. Include the following files if you create your own project:

About32.frm
Main32.frm
Servo32.frm
Step32.frm
VBDemo.bas

..\mcapi32.bas
..\mcdlg32.bas

Set frmMain as the startup object for the project.

Delphi Programming

Included with each of the Delphi program sample (PasDemo) is a read me file (readme.txt) that describes how to build the sample program. The following text was reprinted from the readme.txt file for the PasDemo program sample.

Contents
========

- About the sample
- How to build the sample
- Contacting technical support

About the sample
================

This sample demonstrates a simple user interface to one axis of a motion controller. The user may program moves and interact with the motion in a number of ways (stop it, abort it, etc.). Sample forms demonstrate how to configure servo or stepper motor axes. A number of the new MCDialog functions (such as a full-featured, ready-to-run
axis configuration dialog) are also demonstrated.

How to build the sample
=======================

To build the samples you will need to create a new project or use the Delphi project files included with the sample (Pdemo.dpr for 16-bit, Pdemo32.dpr for 32-bit). Include the following files if you create your own project:

About.pas
Global.pas
PasDemo.pas
Servo.pas
Stepper.pas

For 16-bit projects you will also need:

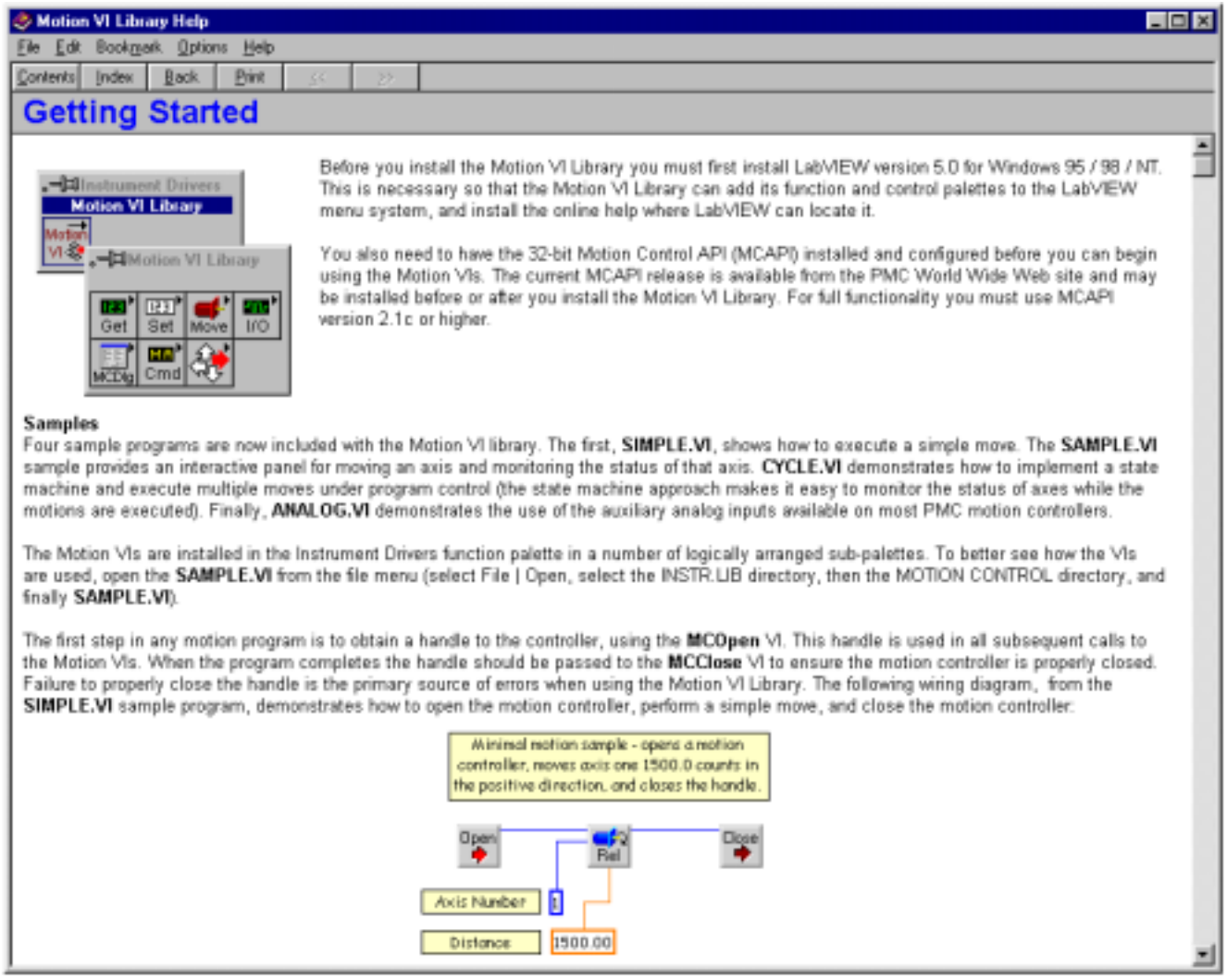
..\mcapi.pas
..\mcdlg.pas

For 32-bit projects you will also need:

..\mcapi32.pas
..\mcdlg32.pas

LabVIEW Programming

PMC's LabVIEW Virtual Instrument Library includes an On-Line help with a Getting Started guide.



PMC Sample Programs

Sample programs with full source code are supplied with the MCAPI. These C++, Visual Basic, and Delphi sample programs allow the user to:

- Move an axis
- Monitor the actual, target, and optimal positions of an axis
- Monitor axis I/O (Limits +/-, Home, Index, an Amplifier Enable)
- Define or change move parameters (Maximum velocity, acceleration/deceleration)
- Define or change the servo PID parameters

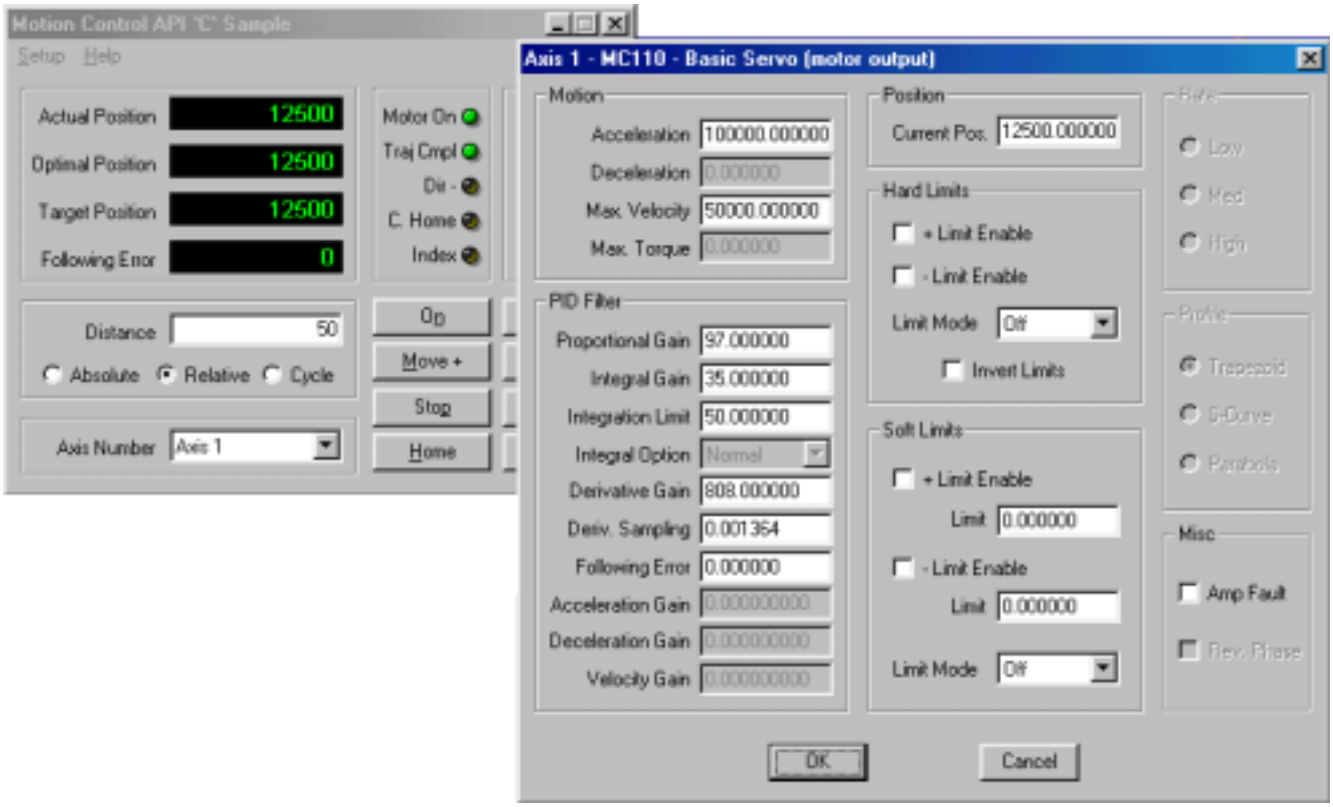


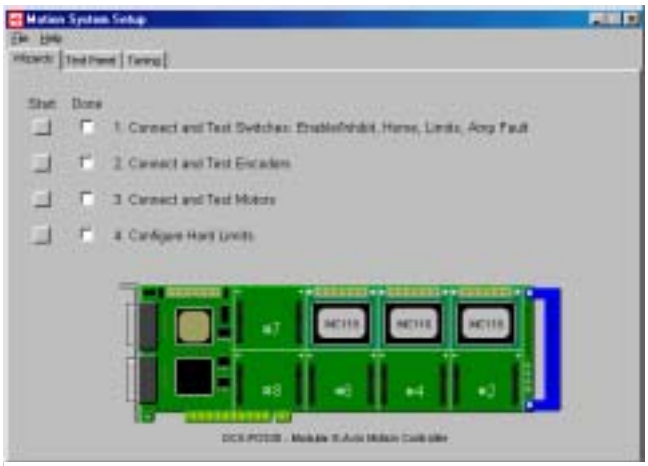
Figure 19: PMC's CWdemo32 includes the executable and source code

Motion Integrator

PMC's Motion Integrator program is just like having your own 'Systems Integrator' to assist you with every step of the integration process. Motion Integrator is a suite of powerful Windows tools that are used to:

- Configure the DCX motion control system
- Verify the operation of the control system
- Execute and plot the results of single and/or multi-axes moves
- Connect and test I/O
  - Axis I/O (Home, Limits, Enable)
  - General purpose Digital I/O
  - General purpose Analog I/O
- Tune the servo axes
- Diagnose controller failures
- View comprehensive on-line help including detailed wiring diagrams

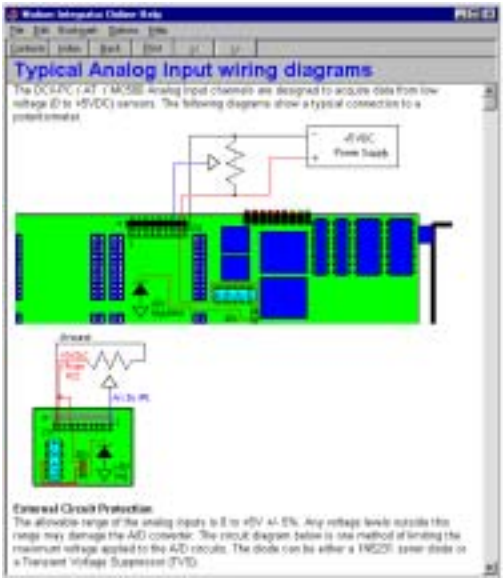
For first time PMC motion control users, Motion Integrator can be run as a series of Windows Wizards



The Motion System Setup program opens with a picture of the DCX controller and a listing of the recommended integration steps

The Axis I/O wizard allows the user to verify the operation of the Limits, Home, and Amp/Drive Enable

Once the systems has been tested and tuned, PMC's Motor Mover allows users to: move any or all motors, define cycling routines, monitor position and status



The on-line help provides detailed information, wiring diagrams, and application examples.

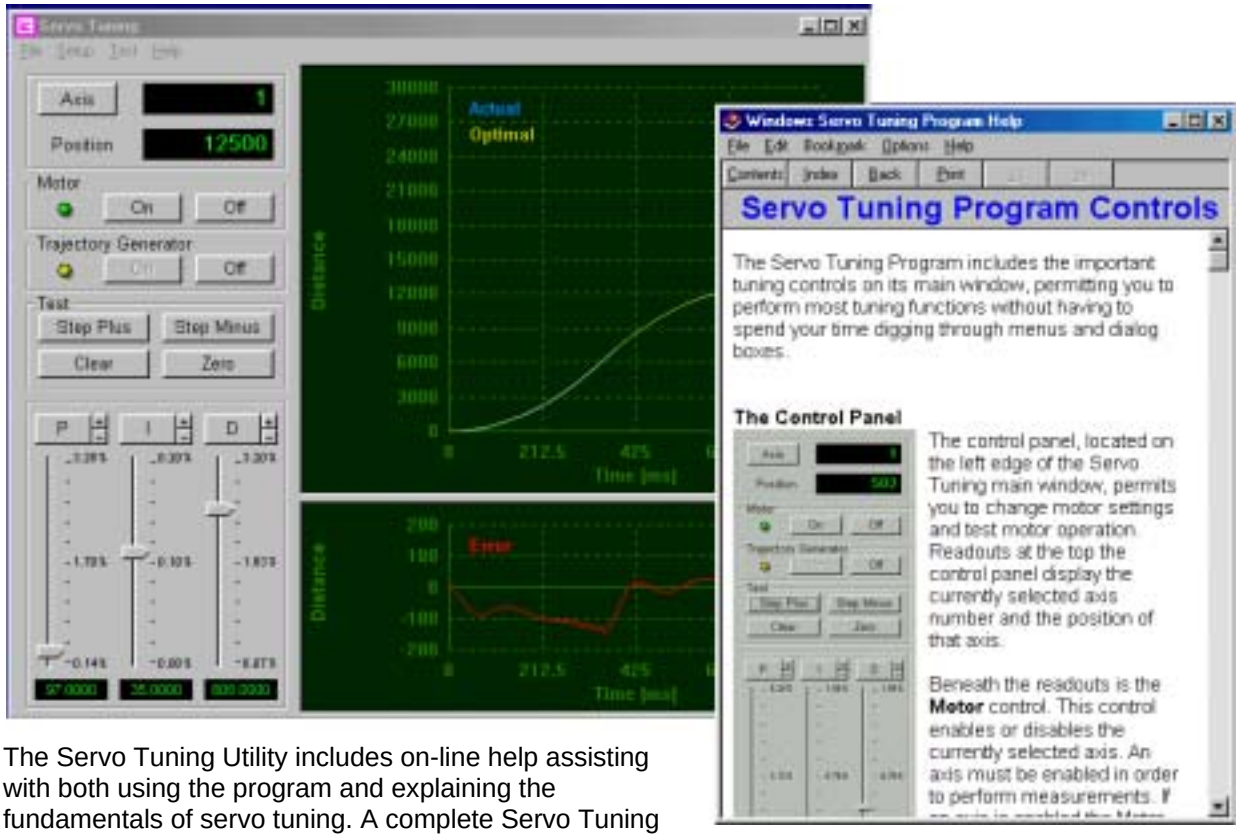
Tuning servo's with Motion Integrator

Motion Integrator provides a powerful and easy to use tool for 'dialing in' the performance of servo systems. From simple current/torque mode amplifiers to sophisticated Digital Drives, Motion Integrator makes tuning a servo is quick and easy.

By disabling the Trajectory generator, the user can execute repeated Gain mode (no ramping - maximum velocity or acceleration/deceleration) step responses to determine the optimal PID filter parameters:

- Proportional gain
- Derivative gain
- Derivative sampling period
- Integral gain
- Integration Limit

With the Trajectory generator turned on, the user can execute 'real world' moves displaying the calculated position, actual position, and following error plots.



The Servo Tuning Utility includes on-line help assisting with both using the program and explaining the fundamentals of servo tuning. A complete Servo Tuning tutorial is available on the MotionCD

Digital and Analog I/O Test Panels

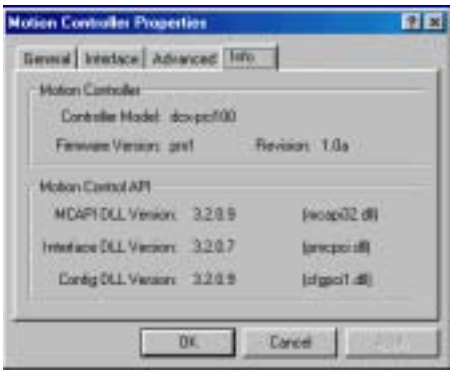
Motion Integrator Digital I/O, and Analog I/O allow the user to verify the operation of general purpose I/O.

PMC Utilities

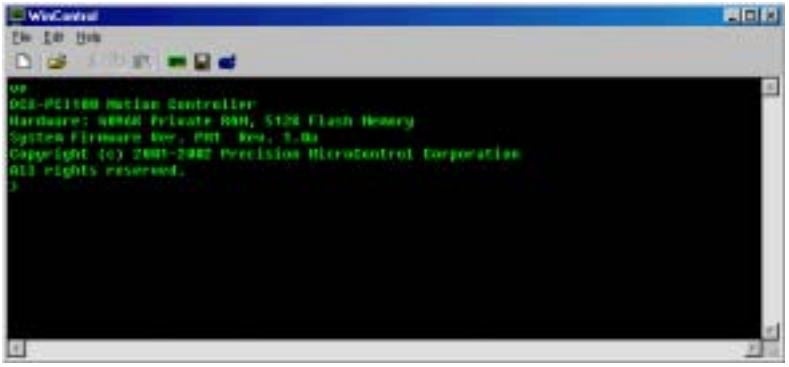
A powerful suite of utilities are included with the Motion Control API. These tools allow the user:

- Query motion control system version information
- Issue native language (MCCL) commands directly to the DCX controller
- Upgrade the firmware of the DCX controller
- Display the position of any or all axes

PMC's Motion Control Panel
The Motion Control Panel is used to query the motion control system for firmware and software (MCAPI) version information, and remove a controller. It can be launched either from the Windows **Start** menu or by selecting the Motion Control icon from the Windows Control Panel.



WinControl – MCCL (Motion Control Command Language) command set interface utility
This utility provides the user with a direct communication interface with the DCX-PCI100 in its native language (MCCL). This tool is extremely useful not only during initial controller integration but also as a debug tool during application software development. Two methods of executing MCCL commands are supported: A PC keyboard key stroke is passed directly to the DCX controller, and/or download a MCCL command text file via the **File** – **Open** menu options



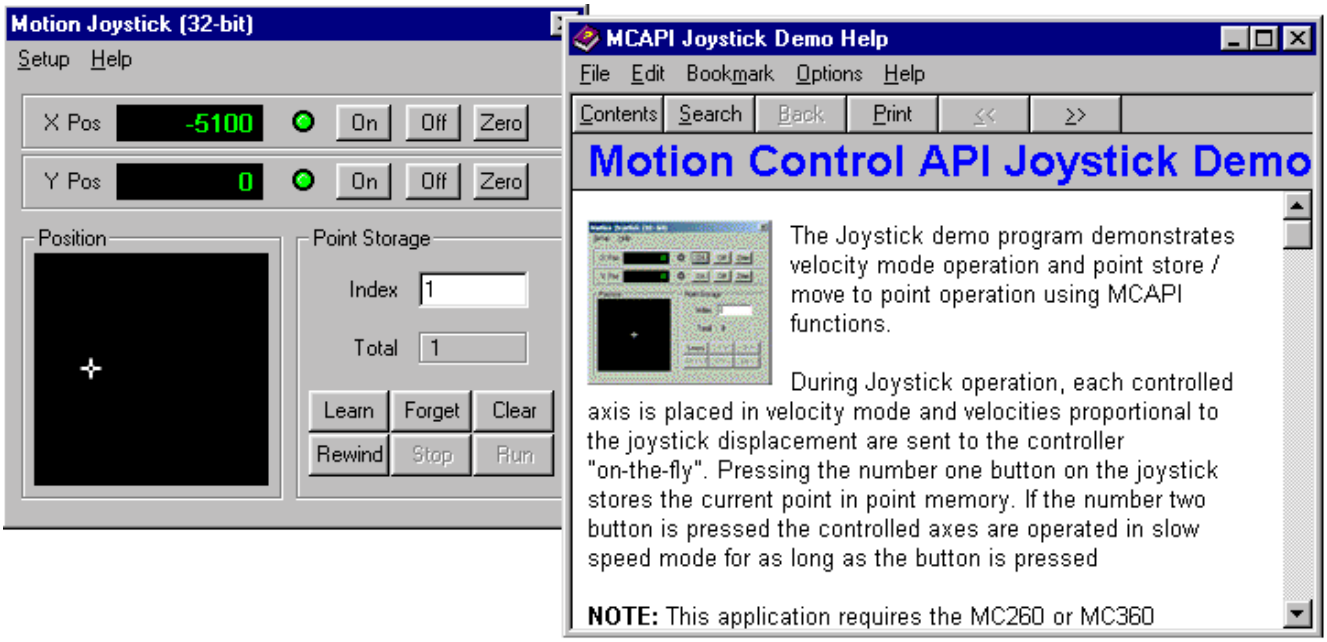
Flash Wizard

To increase CPU efficiency and reduce cost the DCX-PCI100 uses primarily SDRAM. All operational program code (otherwise known as firmware) for the DCX-PCI100 is stored on the hard drive during installation of the MCAPI. When the PC is first powered the MCAPI writes this program code into the on-board SDRAM in a process called Dynamically Loaded Firmware (DLF).

PMC's Flash Wizard is a windows utility that allows the user to easily upgrade the program code from file downloaded from PMC's web site **www.pmccorp.com**.

Joystick Applet

Allows the user to manually position two axes using a joystick connected to the game port of a PC. Full source code for this applet is provided.



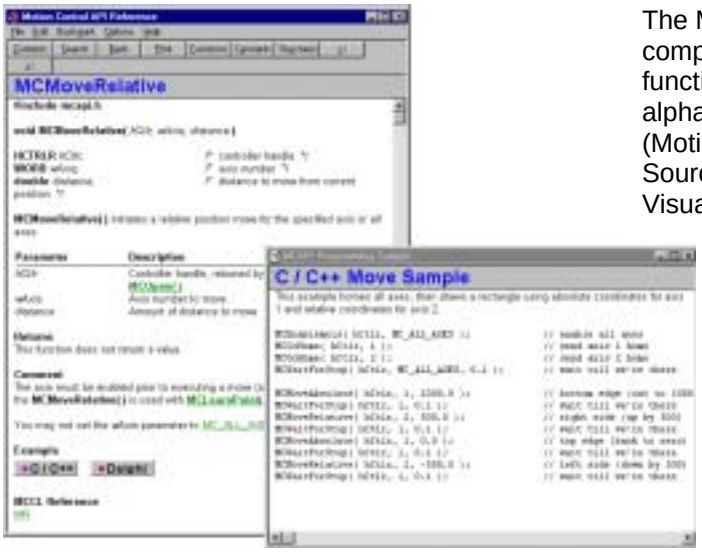
MCAPI On-line Help

Complete and up to date (from PMC website www.pmccorp.com) On-line help for PMC's MCAPI (Motion Control Application Programming Interface). Help documents include; installation and basic usage, complete function call reference and examples, high level dialog descriptions, and LabVIEW VI Library reference.

Mcguide.hlp

The MCAPI Users Guide On-line Help describes the basics of PMC's MCAPI. This should be the '**first stop**' for any questions about the MCAPI.

Mcapi.hlp

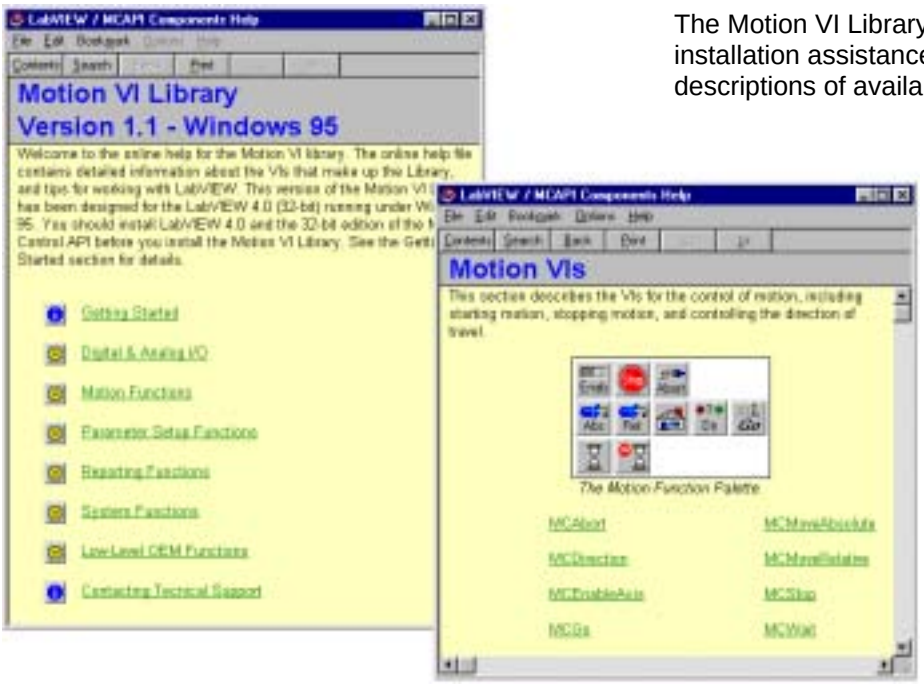


The MCAPI On-line Help provides a complete listing and description of all MCAPI functions. Function calls are grouped both alphabetically and by functional groups (Motion, Setup, Reporting, Gearing, etc...). Source code examples are provided for C++, Visual Basic, and Delphi.

Mcdlg.hlp

The MCAPI Common Dialog On-line Help describes the high level MCAPI Dialog functions. These operations include: Save and Restore axis configurations (PID and Trajectory), Windows Class Position and Status displays, Scaling, and I/O configuration.

Mclv.hlp



The Motion VI Library On-line Help provides installation assistance and detailed descriptions of available VI's.

Chapter Contents

Chapter 5

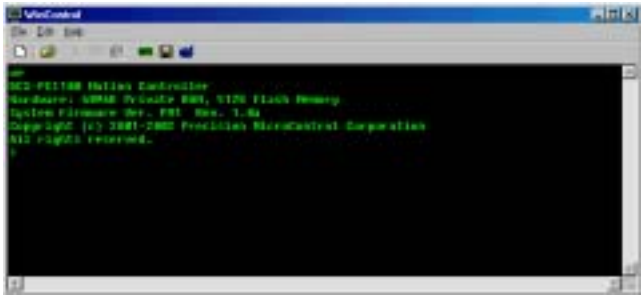
Communication Interfaces

High Speed Binary interface

For PC based application programs the DCX controller provides a high speed binary interface for communicating with the PC via the PCI bus. This interface is implemented using dual ported memory and is mapped into the PC by the BIOS during ‘Plug and Play’ bus enumeration. PMC’s MCAPI provides Windows device drivers and a high level function library for C++, Visual Basic, Delphi, and LabVIEW applications programming. For additional information about available software and integration tools please refer to the **Programming, Software, and Utilities** chapter.

ASCII MCCL Interface

The DCX-PCI100 also provides a PCI ASCII communication interface. When using the WinControl utility the ASCII interface allows the user to communicate directly with the DCX in its native language, MCCL (Motion Control Command Language). The WinControl utility is installed as a component of the MCAPI (Motion Control Application Programming Interface), which is available from PMC’s **Motion CD** or web site **www.pmccorp.com**



In addition to allowing the user to issue MCCL commands from the keyboard one character at a time, the WinControl utility supports downloading a MCCL text file to the controller. Simply store the command lines in a file using a text editor. Use WinControl's File menu option to open the file. Each command line will be executed as it is displayed. Documenting commands can be added to the MCCL program by preceding the comment by a semi colon.

Commands sent to the DCX through any of the ASCII communication interfaces **must be followed by a carriage return (ASCII 13)**. A **linefeed (ASCII 10) is not required** at the end of command lines, and should not be sent.

Chapter Contents

Chapter

6

DCX Operation Basics

Introduction

At its lowest level the operation of the DCX is similar to a microprocessor, it has a predefined instruction set of operations that it can perform. This instruction set, known as MCCL (Motion Control Command Language), consists of over 130 operations that include motion, setup, conditional (If/Then), mathematical, and I/O operations.

However the typical PC based application will never use these low level commands. Instead the programmer will call high level functions (C++, Visual Basic, Delphi, or LabVIEW), which are passed to the DCX via the MCAPI device driver. A example MCAPI function description is:

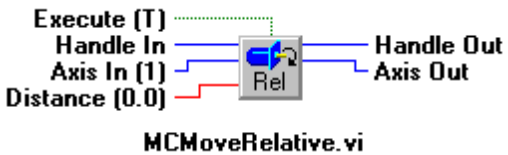
Move to relative position

This command generates a motion of relative *Distance* of n in the specified direction. A motor number must be specified and that motor must be in the on state for any motion to occur. If the motor is in the off state, only its internal target position will be changed.

compatibility: MC100,
see also: Move to absolute position

C++ Function: void MCMoveRelative(HCTRLR hCtlr, WORD wAxis, double Distance);
Delphi Function: procedure MCMoveRelative(hCtlr: HCTRLR; wAxis: Word; Distance: Double);
VB Function: Sub MCMoveRelative (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal distance As Double)
MCCL command: aMRn *a* = Axis number *n* = integer or real

LabVIEW VI:



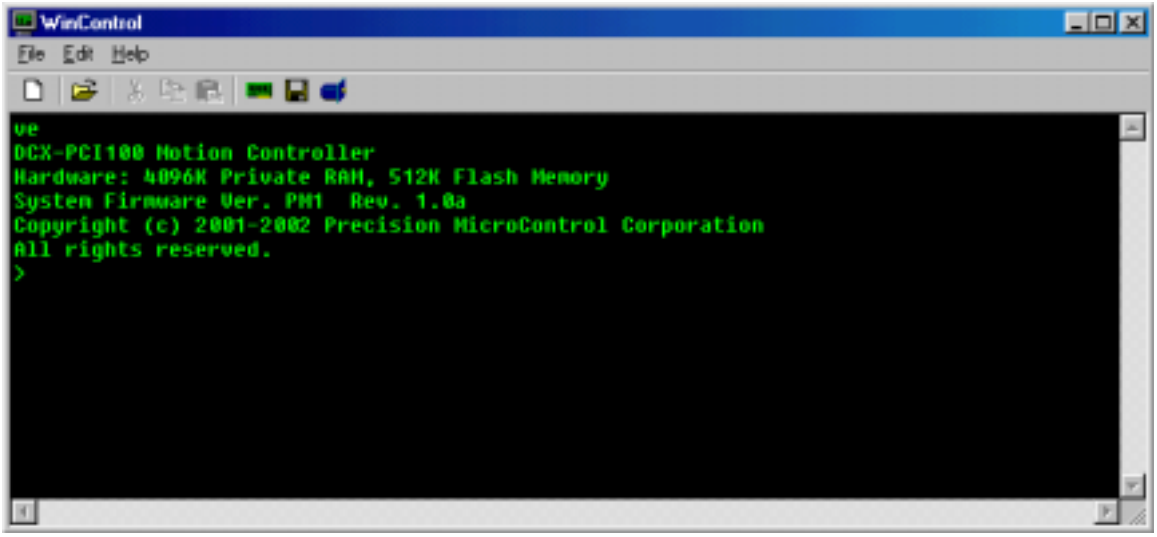
Throughout this manual, when a DCX operation is referenced, the MCAPI command function will be identified by bold, italicized text. The following description differentiates between an absolute and relative move.

Point to Point motion is commanded using either of two DCX functions. To move an axis to an absolute position use the function ***MCMoveAbsolute***. To move an axis a relative distance from the current position use the function ***MCMoveRelative***.

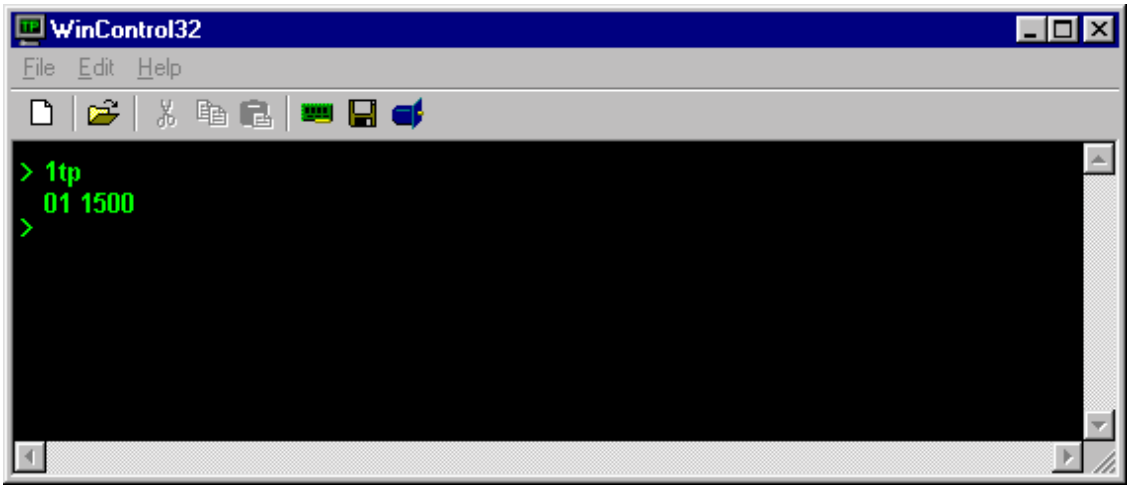
Low Level DCX Operations

The WinControl utility allows the user to communicate with the DCX in the native language (MCCL) of the controller. This utility allows the user to issue MCCL commands directly to the DCX. Each MCCL command is described in detail in the **DCX MCCL Command** chapters later in this user manual.

MCCL commands are two character alphanumeric mnemonics built with two key characters from the description of the operation (eg. "MR" for ***Move Relative***). When the command is received by the DCX (followed by a carriage return) it will be executed. The following graphic shows the result of executing the VE command. This command causes the DCX to report firmware version and the amount of installed memory.



All axis related MCCL commands will be preceded by an axis specifier, identifying to which axis the operation is intended. The graphic below shows the result of issuing the Tell Position (aTP) command to axis number one.



Note that each character typed at the keyboard should be echoed to your display. If you enter an illegal character or an illegal series of valid characters, the DCX will echo a question mark character, followed by an error code. The **MCCL Error Code** listing can be found in the near the end of this manual. On receiving this response, you should re-enter the entire command string. If you make a mistake in typing, the backspace can be used to correct it, the DCX will not begin to execute a command until a carriage return is received.

Once you are satisfied that the communication link is correctly conveying your commands and responses, you are ready to check the motor interface. When the DCX is powered up or reset, each motor control module is automatically set to the "motor off" state. In this state, there should be no drive current to the motors. For servos it is possible for a small offset voltage to be present. This is usually too small to cause any motion, but some systems have so little friction that a few millivolts can cause them to drift in an objectionable manner. If this is the case, the "null" voltage can be minimized by adjusting the offset adjustment potentiometer on the respective module.

Before a motor can be successfully commanded to move certain parameters must be set by issuing commands to the DCX. These include; PID filter gains (servo only), trajectory parameters (maximum velocity, acceleration/deceleration), allowable following error, configuring motion limits (hard and soft).

At this point the user should refer to the **Motion Control** chapter sections titled **Theory of Operation – Motion Control**, and **Servo Operation**. There the user will find more specific information for each type of motor, including which parameters must be set before a motor should be turned on and how to check the status of the axis.

Assuming that all of the required motor parameters have been defined, the axis is enabled with the Motor oN (aMN) command. Parameter ‘a’ of the Motor oN command allows the user to turn on a specific axes or all axes. To enable all, enter the Motor oN command with parameter ‘a’ = 0. To enable a single axis issue the Motor oN command where ‘a’ = the axis number to be enabled.

After turning a particular axis on, it should hold steady at one position without moving. The **T**ell **T**arget (aTT) and **T**ell **P**osition (aTP) commands should report the same number. There are several commands that are used to begin motion, including **M**ove **A**bsolute (MA) and **M**ove **R**elative (MR). To move axis 2 by 1000 encoder counts, enter 2MR1000 and a carriage return. If the axis is in the "Motor oN" state, it should move in the direction defined as positive for that axis. To move back to the previous position enter 2MR-1000 and a carriage return.

With the DCX controller, it is possible to group together several commands. This is not only useful for defining a complex motion, which can be repeated by a single keystroke, but is also useful for synchronizing multiple motions. To group commands together, simply place a comma between each command, pressing the return key only after the last command.

A repeat cycle can be set up with the following compound command:

```
2MR1000,WS0.5,MR-1000,WS0.5,RP6   <return>
```

This command string will cause axis 2 to move from position 1000 to position –1000 7 times. The **R**e**P**eat (RP) command at the end causes the previous command to be repeated 6 times. The **W**ait for **S**top (WS) commands are required so that the motion will be completed before the return motion is started. The number 0.5 following the WS command specifies the number of seconds to wait after the axis has ceased motion to allow some time for the mechanical components to come to rest and reduce the stresses on them that could occur if the motion were reversed instantaneously. Notice that the axis number need be specified only once on a given command line.

A more complex cycle could be set up involving multiple axes. In this case, the axis that a command acts on is assumed to be the last one specified in the command string. Whenever a new command string is entered, the axis is assumed to be 0 (all) until one is specified.

Entering the following command:

```
2MR1000,3MR-500,0WS0.3,2MR1000,3MR500,0WS0.3,RP4   <return>
```

will cause axis 2 to move in the positive direction and axis 3 to move in the negative direction. When both axes have stopped moving, the WS command will cause a 0.3 second delay after which the remainder of the command line will be executed.

After going through this complex motion 5 times, it can be repeated another 5 times by simply entering a return character. All command strings are retained by the controller until some character other than a return is entered. This comes in handy for observing the position display during a move. If you enter:

```
1MR1000   <return>
1TP    <return>
(return)
(return)
(return)
(return)
```

The DCX will respond with a succession of numbers indicating the position of the axis at that time. Many terminals have an "auto-repeat" feature, which allows you to track the position of the axis by simply holding down the return key.

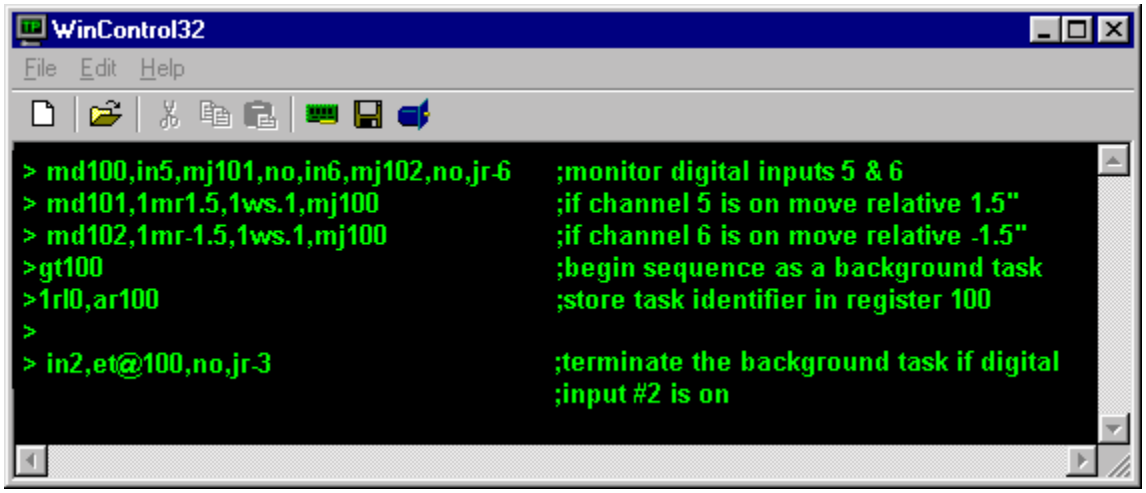
Another way to monitor the progress of a movement is to use the ***Repeat*** command without a value. If you enter:

```
1MR10000  <return>
1TP,RP    <return>
```

The position will be displayed continuously. These position reports will continue until stopped by the operator pressing the Escape key.

While the DCX is executing commands, it will ignore all alphanumeric keys that are pressed. The user can abort the commands by pressing the escape key. If the user wishes only to pause the execution of commands, the user should press the space bar. In order to restart command execution press the

space bar again. If after pausing command execution, the user decides to abort execution, this can be done by pressing the escape key.



Chapter Contents

Chapter 7

Motion Control

This chapter describes the basic building blocks of DCX motion control.

Theory of DCX Motion Control

The DCX motherboard (DCX-PCI100) uses a 192 MHz 32 bit MIPS processor that is programmed to perform motion control tasks. Specially designed servo control modules are installed on the motherboard to configure it for controlling from 1 to 8 servo motors. Each DCX motion control module (DCX-MC100, DCX-MC110) installed on the motherboard provides all the circuitry required to control one motor and its associated axis I/O (home, limits, amp/driver enable, fault, etc...).

Servo Motor Control

The DCX servo modules use a position feedback loop to control the servo. The **DCX-MC100** controls the operation of servo motor via a 12 bit, +/-10 volt analog output signal to an external servo amplifier. The **DCX-MC110** provides a 0 - +12 volt, 8 bit, direct motor drive output capable of directly driving a 12 volt motor with up to 0.5A of current.

Incremental encoder input to these modules provide feedback information for closing the position loop. In operation, the servo module subtracts the actual position (feedback position) from the desired position (trajectory generator position), and the resulting position error is processed by the digital filter on the module. The output of the digital filter sets the module's servo command output level.

The module processor monitors the motor's position via an incremental encoder. The two quadrature signals from the encoder are used to keep track of the absolute position of the motor. Each time a logic transition occurs at one of the quadrature inputs, the DCX position counter is incremented or decremented accordingly. This provides four times the resolution over the number of lines provided by the encoder. The encoder interface is buffered by a differential line receiver on the DCX module. Jumpers on the DCX module allow the user to configure the differential receiver for use with single ended or differential encoder.

A "Proportional Integral Derivative" (PID) digital filter on the module is used to compensate the servo feedback loop. The motor is held at the desired position by applying a restoring force to the motor that

is proportional to the position error, plus the integral of the error, plus the derivative of the error. The following discrete-time equation illustrates the control performed by the servo controller:

u(n) = Kp*E(n) + Ki sum E(n) + Kd[E(n') - E(n' - 1)]

where u(n) is the module's output signal output at sample time n, E(n) is the position error at sample time n, n' indicates sampling at the derivative sampling rate, and kp, ki, and kd are the discrete-time filter parameters loaded by the users. The first term, the proportional term, provides a restoring force proportional to the position error. The second term, the integration term, provides a restoring force that grows with time. The third term, the derivative term, provides a force proportional to the rate of change of position error. It provides damping in the feedback loop. The sampling interval associated with the derivative term is user-selectable; this capability enables the servo controller to control a wider range of inertial loads.

DCX Servo Basics

The basic steps required to implement closed loop servo motion are:

- Proper encoder operation
- Setting the allowable following error
- Verify proper motor/encoder phasing
- Tuning the servo (PID)

Quadrature Incremental Encoder
All closed loop servo systems require position or velocity feedback. These feedback devices output signals that relay position and/or velocity with which motion controller ‘closes the loop’. The most common feedback device used with intelligent motion control systems is quadrature incremental encoder.

A quadrature incremental encoder is an opto electric feedback device. A light source and photo sensor pickup are used to detect markings on a glass ‘scale’. The more markings on the glass scale, the higher the resolution of the encoder. Circuitry connected to the photo sensor generates two wave forms (Phase A and Phase B), which have a phase difference of 90 degrees. This phase difference is used by the encoder input circuitry of the DCX to:

Determine the direction of rotation (positive or negative) of the encoder/motor
Enhance the resolution of the encoder by a factor of 4.

For example, a 500 line quadrature incremental encoder will have 2000 encoder counts per full rotation. The 90 degree phase difference is also used to determine the direction of motion of the encoder. If phase A comes before phase B, the DCX will determine that motion is in the positive or clockwise direction. If phase B comes before phase A, the DCX will determine that motion is in the negative or counter-clockwise direction.

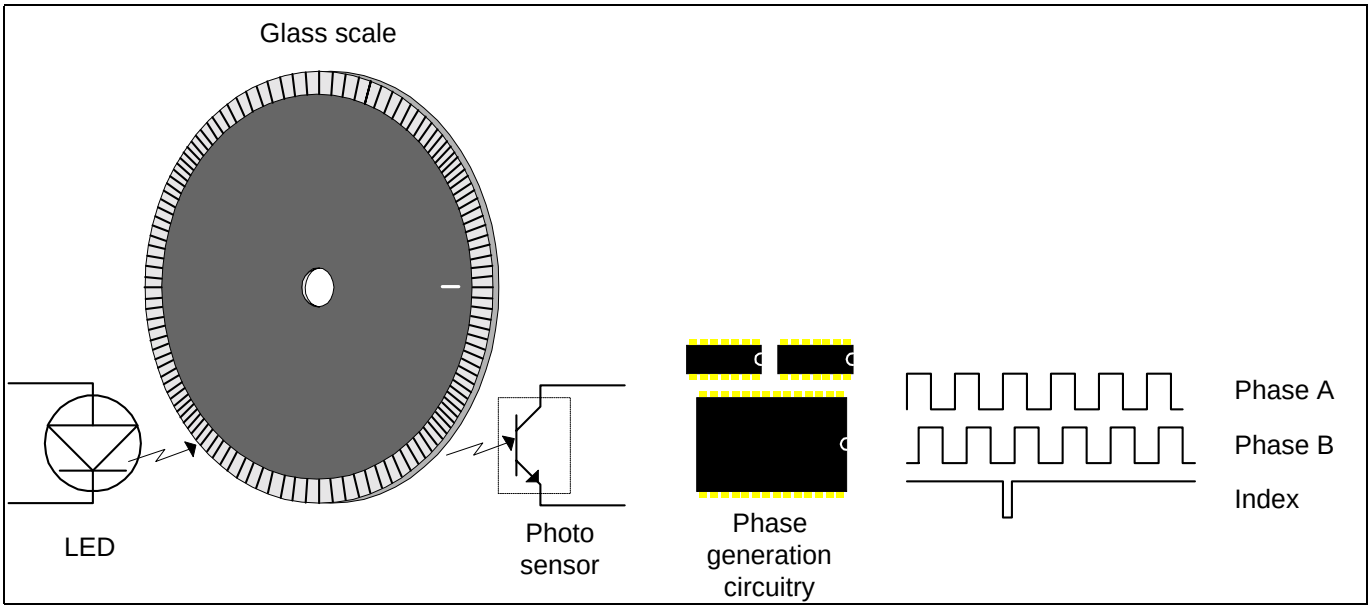
Some quadrature encoders include an additional ‘mark’ on the glass scale that is used to generate an index pulse. This signal, which ‘goes active’ once per rotation, is used by the motion controller to accurately home (re-define the position of an axis) the axis. Please refer to the **Homing Axes** section of this chapter.

There are few options that are typically associated with quadrature encoders.

Output type: Differential or single ended
Differential outputs (A+, A-, B+, B-) are recommended for superior noise immunity but the DCX supports either output type

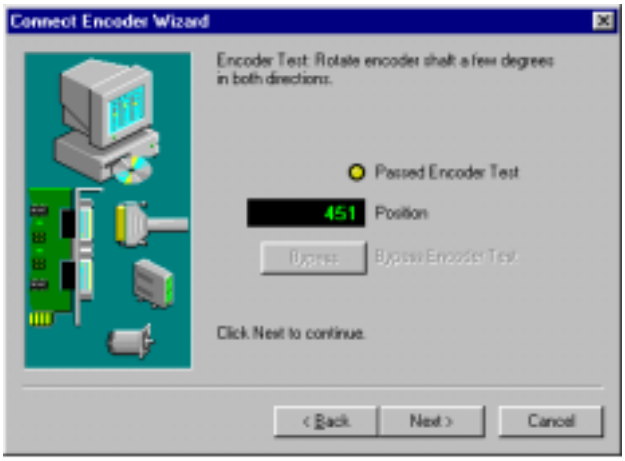
Index or no Index (used for homing the axis)
MC100/110 modules support only Z-. For Differential Index (Z+, Z-) the DCX-BF100 interconnect assembly is required.

+5 volt supply required or +12 volt supply required.
A +5 volt encoder is recommended but the DCX also supports a +12V encoder

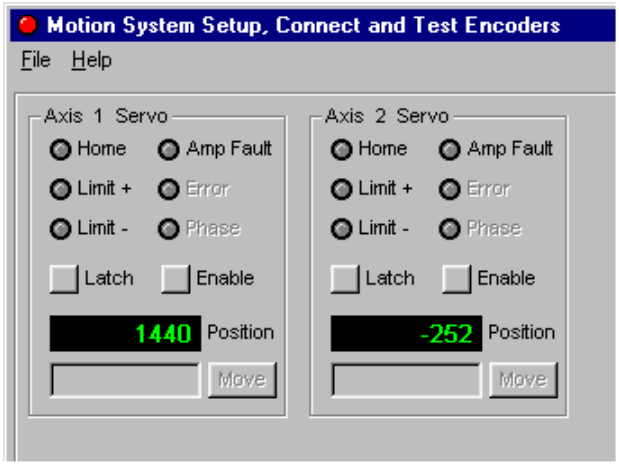


Encoder Checkout
The Motion Integrator program provides easy to use tools for testing the operation of an encoder.. The user has the option of using the Connect Encoder Wizard or the Motion System Setup Test Panel.

Note – Unlike the Connect Encoder Wizard, the Motion System Setup Test panel **does not** allow the user to verify the operation of the encoder Index.



Motion System Setup Connect Encoder Wizard



Motion System Setup Motor Test Panel

Manually rotate the motor/encoder in either direction, the position reported should increment or decrement accordingly. Refer to the Troubleshooting guide if the DCX does not report a change of position.

Setting the Allowable Following Error

Following error is the difference between where an axis **'is'** and where the controller has **'calculated it should be'**. All servo systems require 'some' position error to generate motion. When a servo axis is turned on, if a position error exists, the PID algorithm will cause a command voltage to be applied to the servo to correct the error.

While an axis is executing a move, the following error will typically be between 20 and 100 encoder counts. Very high performance systems can be 'tightly tuned' to maintain a following error within 5 to 10 encoder counts. Systems with low resolution encoders and/or high inertial loads will typically maintain a following error between 150 and 500 encoder counts during a move.

The DCX supports 'hard coded' following error fault checking (which by default is disabled, allowable following error = 0). To enable following error checking set the allowable following error to a non zero value between 1 and 32767. after making this change if at anytime the difference between the optimal position and the current position exceeds the user defined 'allowable following error', an error condition will be indicated. The axis will be disabled (Amplifier Inhibit output turned on, output command signal set to 0.0V) and the axis status word will indicate that an Motor Error has occurred. The ***MCEnableAxis( )*** function is used to clear a following error condition. The following error fault checking cannot be disabled, the maximum allowable following error is 32767 encoder counts.

The three conditions that will typically cause a following error fault are:

1) Improper servo tuning (Proportional gain **too low**)
2) Velocity profile that the system cannot execute (moving too fast)
3) The axis is reversed phased (move positive causes encoder position to begin decrementing)

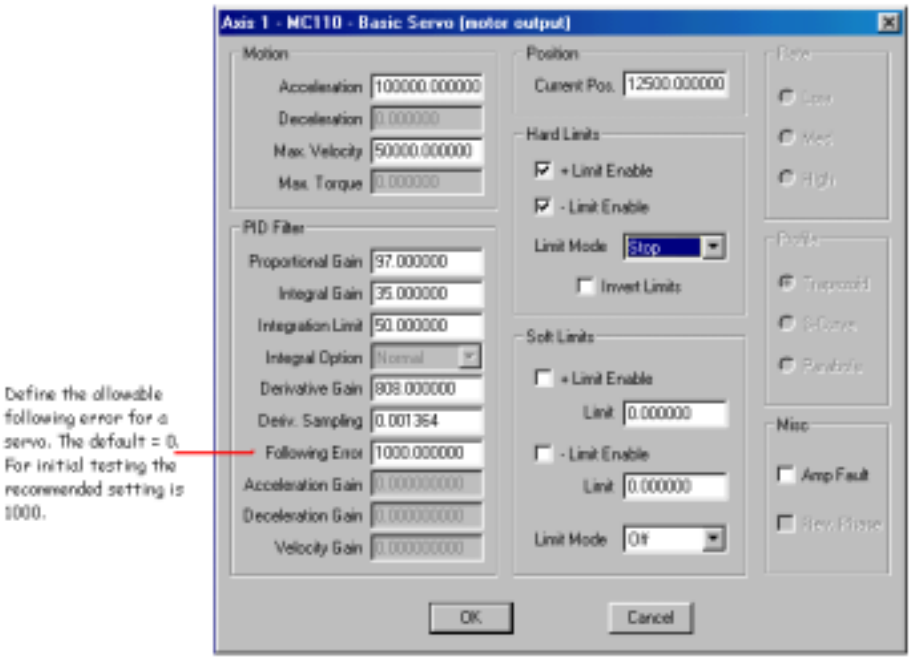
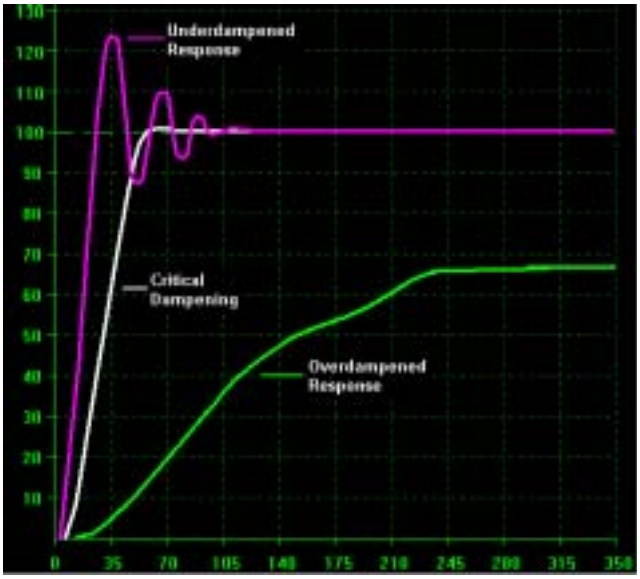


Figure 20: From Servo Tuning or Motor Mover use the Servo Dialog box to redefine the allowable following error

Tuning the Servo

A servo motor motion system is a closed loop system with negative feedback. Servo tuning is the process of adjusting the gains (proportional, derivative, and integral) of this axis controller to get the best possible performance from the system. A servo motor and its load both have inertia, which the servo amplifier must accelerate and decelerate while attempting to follow a change in the input (from the motion controller). The presence of inertia will tend to result in over-correction, with the system oscillating or "ringing" beyond either side of its target (under-damped response). This ringing must be damped, but too much damping will cause the response to be sluggish (over-damped response). Proper balancing will result in an ideal or critically-damped system.



The servo system is tuned by applying a command output or ‘step response’, plotting the resulting motion, then adjusting parameters of the digital PID filter until an acceptable system response is achieved. A step response is an output command by the motion controller to a specific position. A typical step response distance used for tuning a servo is 100 encoder counts. If the system requires:

- Very short duration moves (less than 100 msec’s)
- Very small following error value (less than 20 encoder counts

Then a step response of 50 encoder counts is recommended. If the servo system is moving a high inertial load (minimal friction) then the step response should be increased to 200 – 300 encoder counts. There is a ‘loose’ relationship between the step response and the following error of the system. The shorter the step response when tuning the servo, the lower the following error during application motion.

Note – Using an ultra short step response (5 – 20 counts) may result in an unstable system that oscillates during and after a commanded move.

During Servo Tuning the DCX-PCI100 will perform one Motion Data Capture operation every millisecond. If more than one DCX motor module is installed, the period between data captures for the target axis will be:

1 msec. X # of installed modules

For example if 6 motor modules are installed, and the MCCaptureData function is called for axis #1, motor data will be captured for axis #1 every 6 msec's.

1 msec. X 6 modules = 6 msec's

Tuning Step #1 - Open the Servo Tuning Utility (Start\Programs\Motion Control\Motion Integrator\Servo Tuning). From the menu bar select **Setup** and then **Test Setup.** Configure the Test Setup dialog as shown (commanding a 100 encoder count step response with display window period set to 500 msec's):

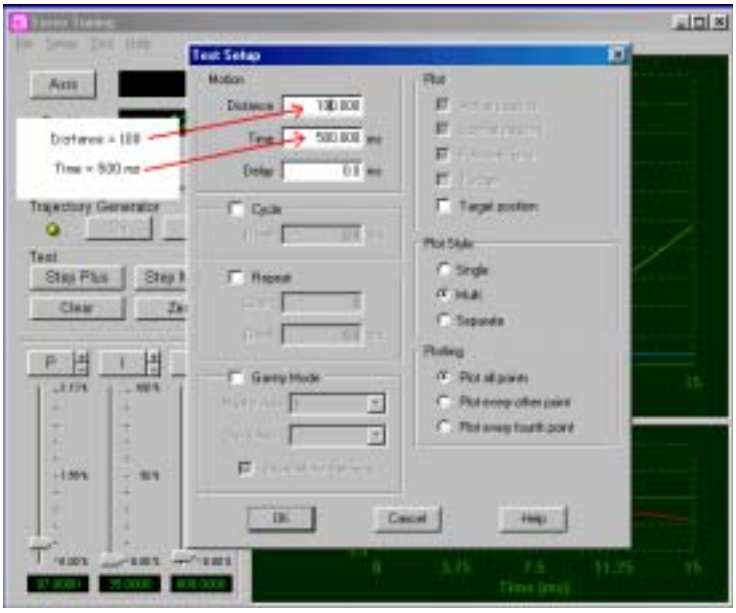
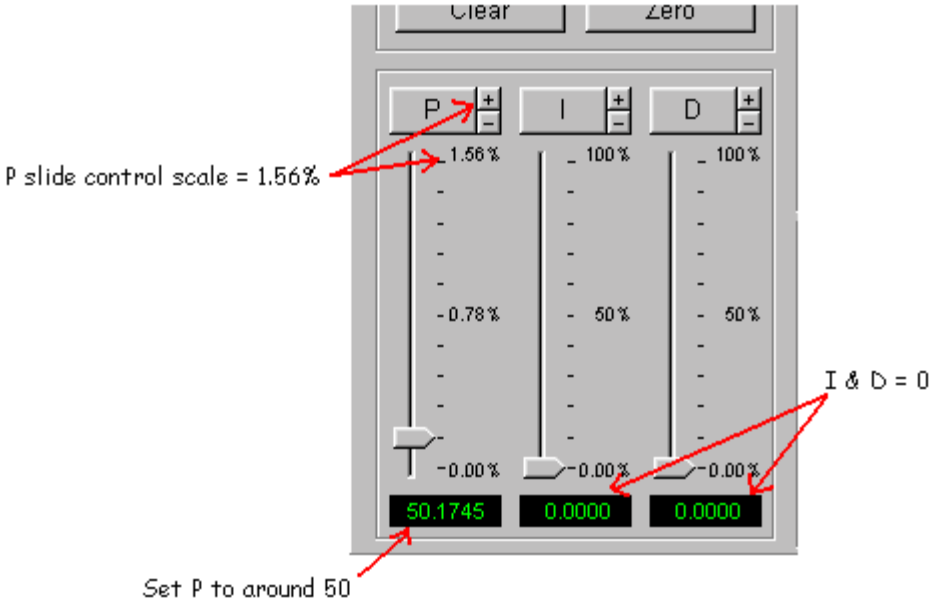
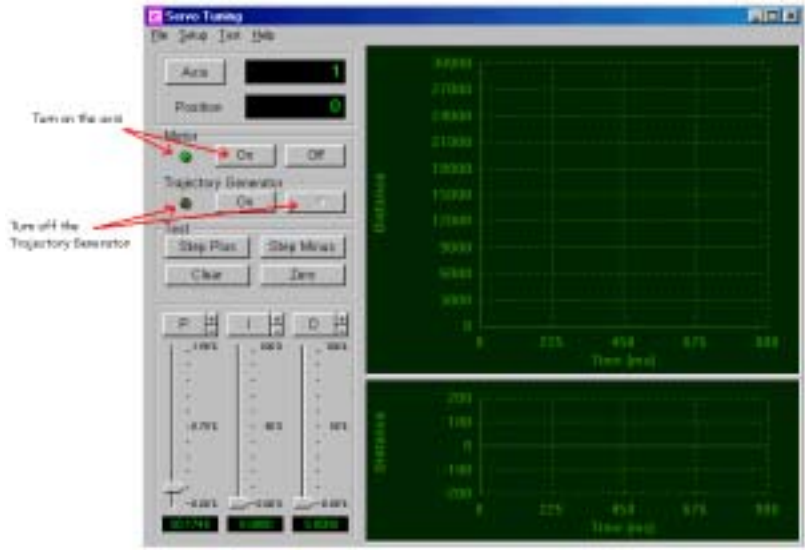


Figure 21: Set Step Distance to 100 encoder counts and Time period to 500 miliseconds

Tuning Step #2 - Verify that the I & D slide controls are all the way down (set to 0). Select the P 'zoom in' (+) button until the scale display is set to 1.56%. Set the P slide control to a value of approximately 50.

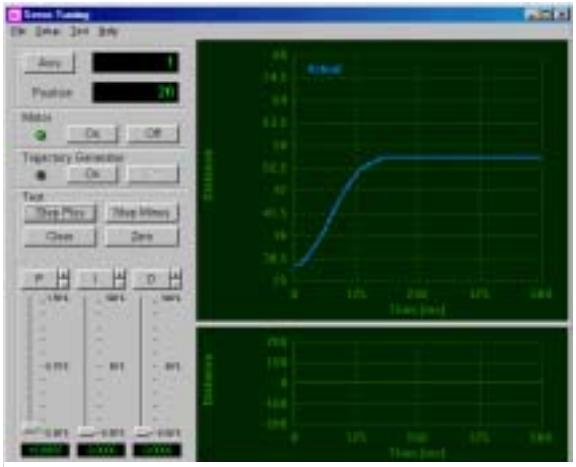


Tuning Step #3 - Turn on the axis and turn off the Trajectory Generator. While setting proportional and derivative gain, the step response should occur with the **Trajectory Generator** disabled. This will result in the magnitude of the output signal being determined only by a PD filter, the controller will not apply a maximum velocity or ramping (acceleration/deceleration).



Tuning Step #4 - Find the Proportional gain value that causes the axis to cross the **target 3 times (no more and no less)**. Before each move press the Clear and Zero buttons to initialize the display and the position of the axis. To move select either the Step+ or Step - buttons. If the proportional gain is too low the axis may:

Not move at all
It may move but not reach the target
It may reach the target but not cross three times



Doesn't reach target - P too low

Crosses target only once - P too low

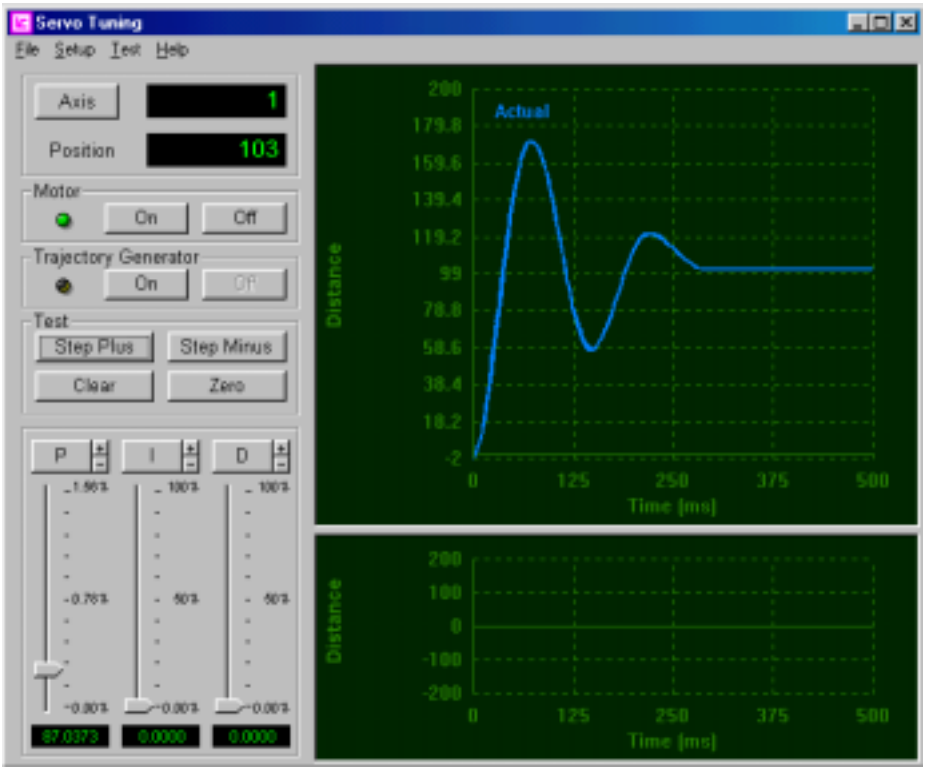
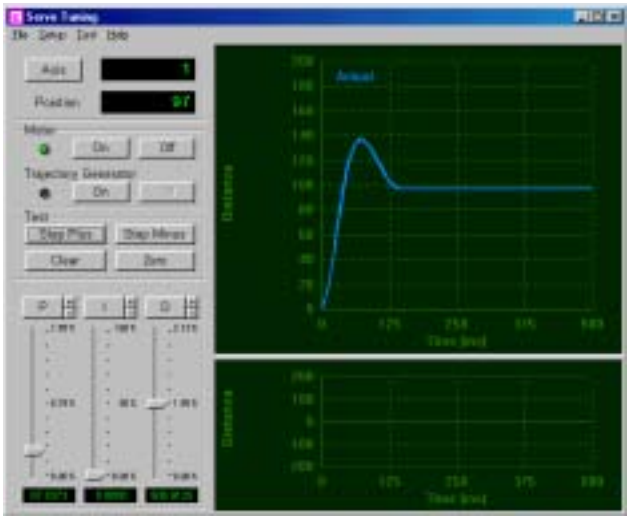


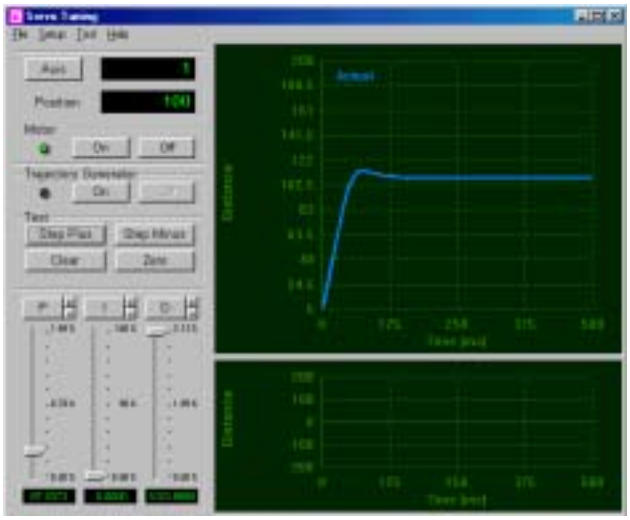
Figure 22: Axis crosses the target 3 times - good setting for proportional gain

If no plotted position path is shown and the ***Motor On LED is off*** an error has occurred. The most likely cause is a following error, indicating that the servo is reversed phased. Open the Servo Setup dialog box and select the Reverse Phase option or 'swap' the phase A and B connections from the encoder to the DCX servo module. Turn the motor back on and proceed with the tuning process.

Tuning Step #5 - Derivative gain dampens the response of the servo system. In this step the goal is to limit the overshoot of a step response to no more than 25%. In the last step response the maximum position of the axis is approximately 160 counts (an overshoot or 60%). Increase the derivative gain until the maximum position is no greater than 125 counts. Before setting the derivative gain you must first set the Derivative Sampling period. The derivative sampling period is expressed in servo loop periods (0.000341 micro seconds). For a typical servo system set the derivative sampling period to 0.000682 seconds (2 loop periods). For a high inertia servo system set the derivative sampling period to 0.001354 seconds (4 loop periods). For a high friction servo system set the derivative sampling period to 0.000341 seconds (1 loop period). Set the D slide control scale to 3.13% by repeatedly pressing the D + button. Set the D slide control to approximately 50% and execute a step response.



Derivative gain setting of 508 limits overshoot to around 40% - the servo is under dampened, increase Derivative gain



Derivative gain setting of 1023 limits overshoot to around 10% - the servo is over dampened, decrease Derivative gain

Figure 23: Derivative gain setting of 789 limits overshoot to 25% - good setting for derivative gain

A general guideline for the derivative gain is that it should not be more than 10 times greater than the setting of proportional gain. If the derivative gain is 10 times greater than proportional gain double the Derivative Sampling setting.

Tuning Step #6 - Setting the Integral gain. Due to friction, 'sticktion', amplifier offset, etc... most servo systems are unable to settle at the target if using only proportional and derivative gain. Integral gain provides a restoring force that increases with time. It is used to correct a static position error of a servo system. If the servo is unable to repeatedly position within +/- one encoder count of the target Integral Gain will, in most cases, position the servo at the target. To configure the Servo Tuning utility for setting the integral gain:

- Enable the trajectory generator.
- Define trajectory parameters (max. velocity, accel / decel) in the Servo Setup dialog
- Define a typical application move distance and duration in the Test Setup dialog
- Set the Integration Limit (typically set to 50)

For this example:

- Maximum velocity = 50,000 counts per second

- Acceleration / deceleration = 100,000 counts per second per second
- Move distance = 12,500 counts
- Plot window time = 700 msec's

With the trajectory generator enabled, a step response will cause two plot traces to be displayed in the upper window and one trace plot in the lower window. The blue trace is a plot of the actual positions of the servo. The yellow trace is a plot of the calculated (or optimal) positions of the servo. The optimal positions are the result of calculations by the DCX based on the trajectory parameters (max. velocity, accel / decel) defined in the Servo Setup dialog. The red trace is a plot of the following error (the difference between the calculated positions and the actual positions. With no integral gain setting a typical system response would be:

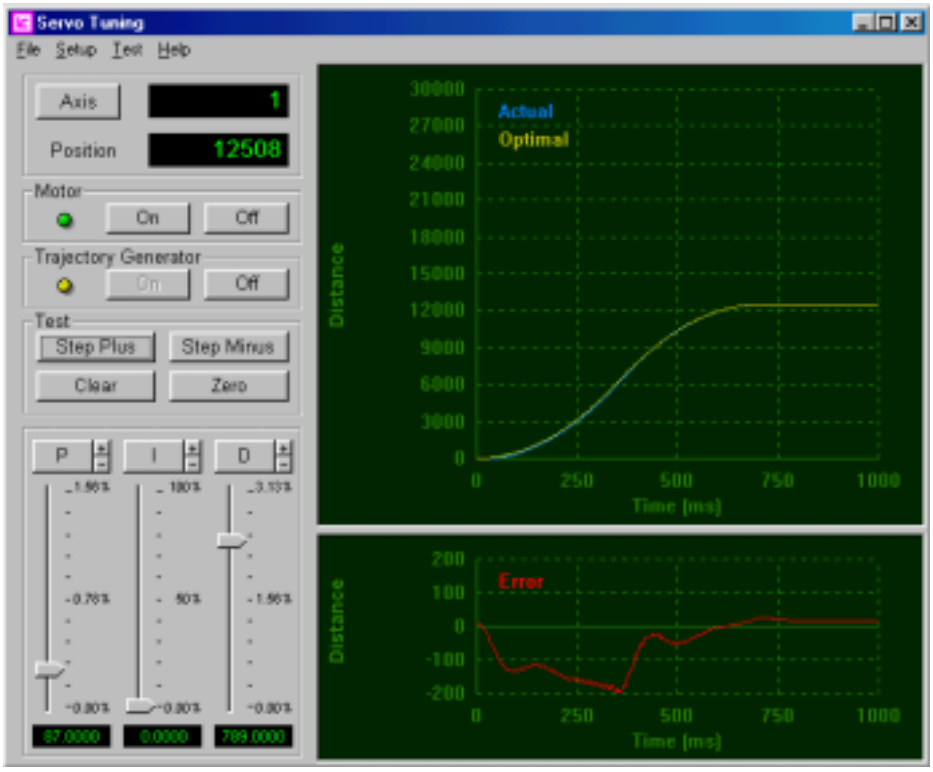


Figure 24: Without Integral gain the axis is 8 counts from the target

Set the I slide control scale to 0.78% by repeatedly selecting the I zoom in (+) button. Without executing another move, slowly increase the integral gain (I slide control) until the position readout indicates that the axis has reached the target position of the move.

Now repeat the move, if the axis settles within one encoder count the axis has been tuned. If the axis fails to settle (position changing) reduce the integral gain setting and repeat the move.

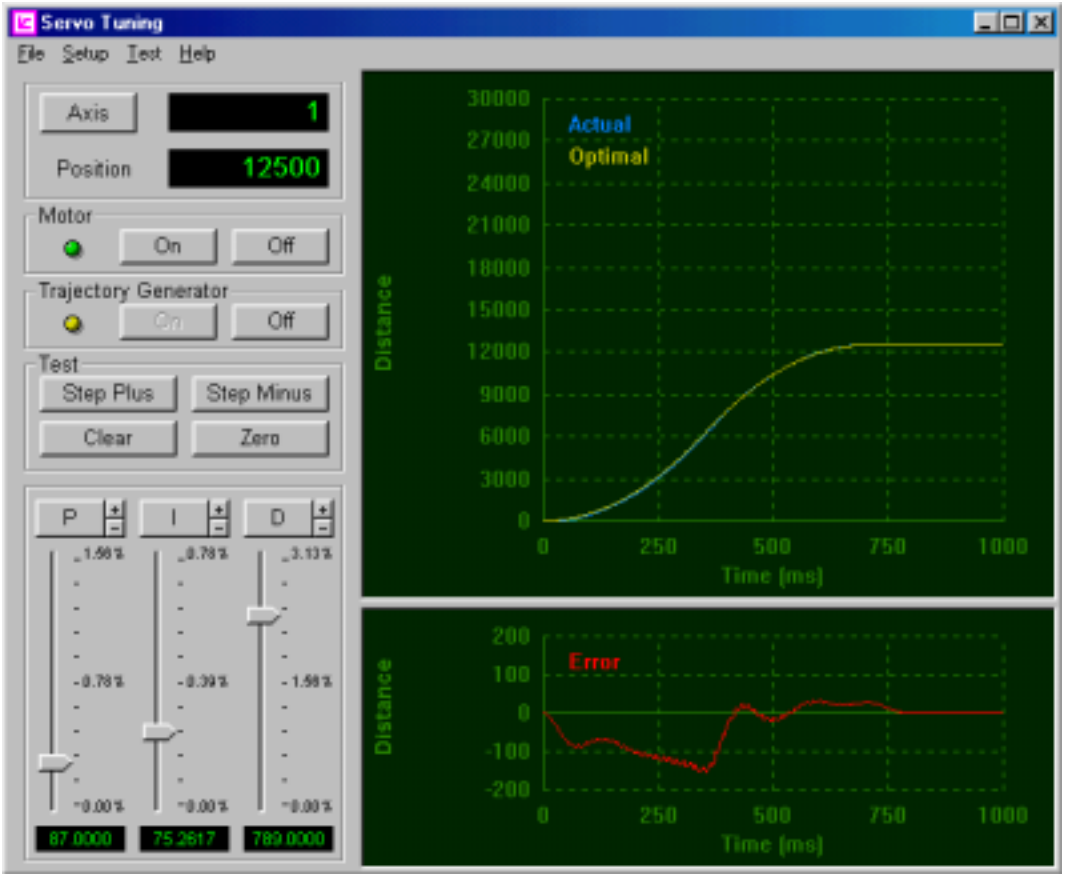
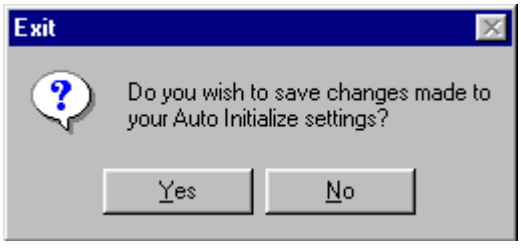


Figure 25: Tuning is complete, axis stops and settles within 1 encoder count

If the Integral gain setting exceeds 200 and is still more than 2 counts from the target at the end of the move then double the Integration Limit setting.

Tuning Step #7 Saving the Tuning Parameters. When servo tuning is complete, closing the tuning utility will prompt this message about saving the Auto Initialize settings, selecting **Yes** will store all settings for all installed axes in the MCAPI.INI file (in the Widows folder). Selecting **No** will cause all settings to be discarded.

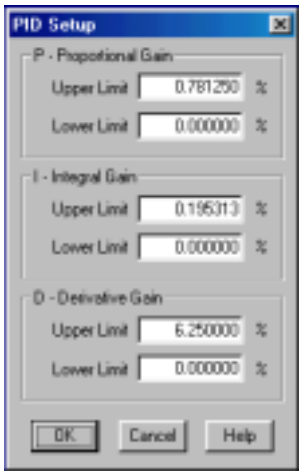


Electing to save the Auto Initialize settings causes the Servo Tuning utility to call the MCAPI Common Dialog function **MCDLG_SaveAxis**. All servo parameters (PID, Trajectory, Limits, etc...) will be saved in the dialog

To define these servo parameters from a user's application program, call the MCAPI Common Dialog function **MCDLG_RestoreAxis**.

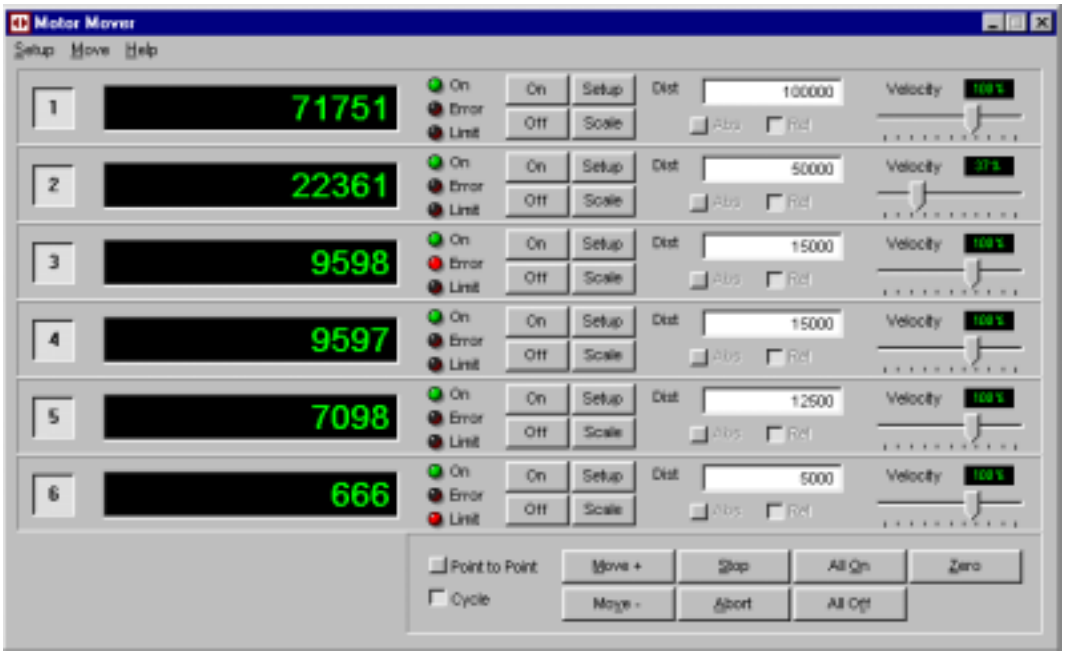
Changing the Scale of the Slide Controls

At the bottom of each slide control is a value showing the current setting as a percentage of the current maximum setting. To change the range of one or more slide controls, using the **Setup Menu**, open the **PID Setup** dialog box (**Setup** – **PID Setup**).



Moving Motors with Motor Mover

After tuning the servo, and setting the trajectory parameters (Max. velocity, accel / decel) the axis is ready to execute motion. The Motor Mover program (Start\Programs\Motion Control\Motion Integrator\Motor Mover) allows the user to execute absolute, relative, and cycle move sequences, monitor position and status of the axis. By selecting the **Setup** button the user can; change velocity parameters (maximum velocity, acceleration/deceleration), PID parameters, and enable motion limits.

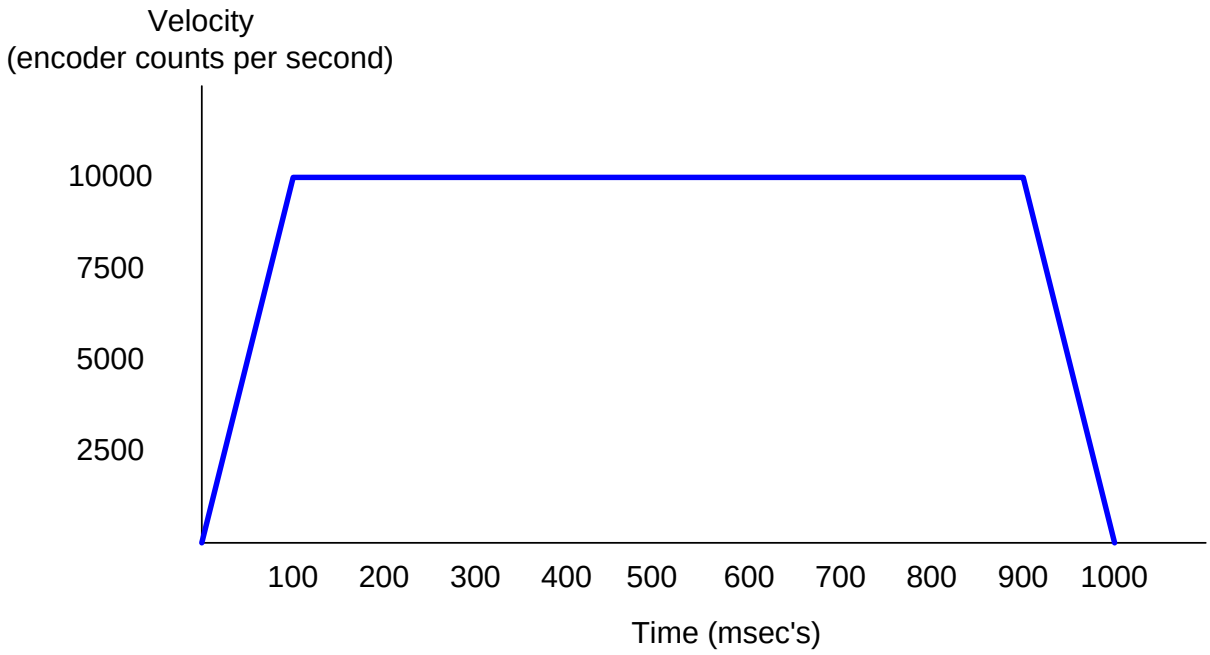


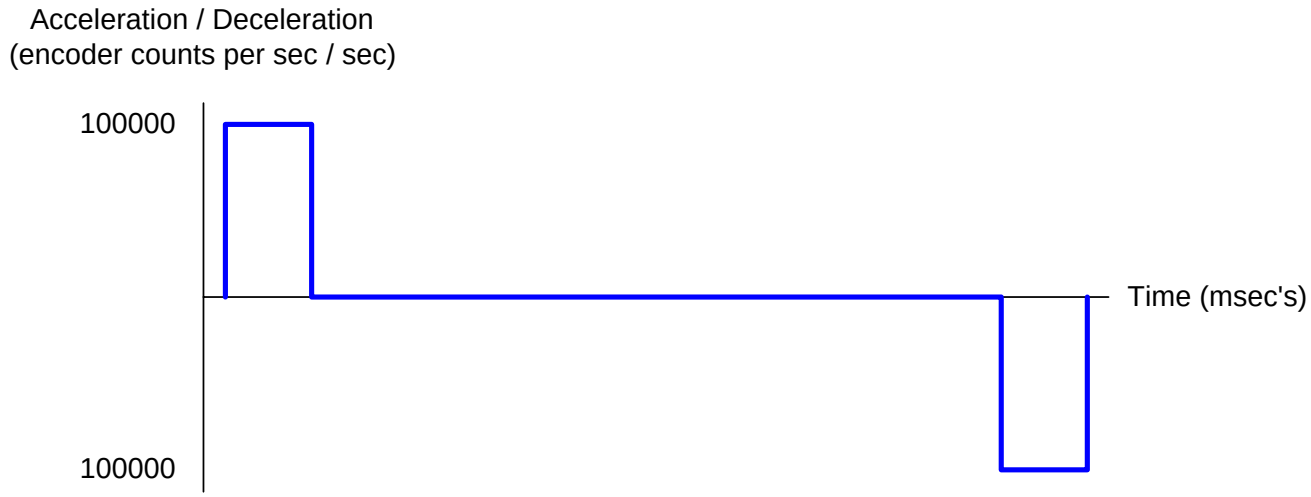
Defining the Characteristics of a Move

Prior to executing any move, the user should define the parameters of the move. The components that make up a move are:

```
// Set axis 1 maximum velocity
// Set axis 1 acceleration/deceleration
// Set Position mode
// Set target (10000), begin move


MCSetVelocity( hCtlr, 1, 10000.0 );
MCSetAcceleration( hCtlr, 1, 100000.0 );
MCSetOperatingMode( hCtlr, 1, 0, MC_MODE_POSITION );
MCMoveRelative( hCtlr, 1, 100000.0 );
```

The parameters defined in the program example above specify a move to position 100,000. During the move the velocity will not exceed 10,000 encoder counts per second. A trapezoidal velocity profile will be calculated by the DCX. The rate of change (acceleration and deceleration) will be 100,000 encoder counts per second/per second, there by reaching the maximum velocity (10,000 counts per second) in 100 msec's. The resulting velocity and acceleration profiles follow:

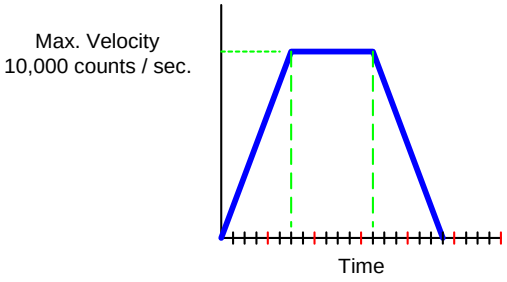




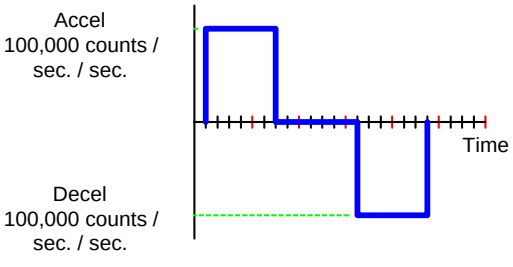
Velocity Profile

The DCX-PCI100 uses a Trapezoidal Velocity Profile to calculate the trajectory of a move.

DCX-PCI100 Velocity Profile



DCX Accel / Decel Profiles



Point to Point Motion

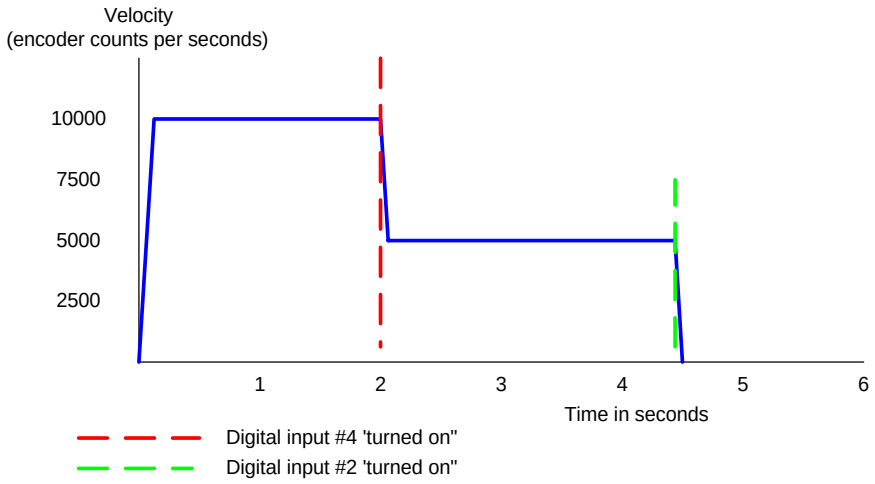
To perform point to point motion of a servo the following steps are required:

```
// Enable the axis
// Enable Position mode
// define maximum velocity
// define acceleration/deceleration
// execute the move

MCEnableAxis( hCtlr, 1, TRUE );
MCSetOperatingMode( hCtlr, 1, 0, MC_MODE_POSITION );
MCSetVelocity( hCtlr, 1, 10000.0 );
MCSetAcceleration( hCtlr, 1, 25000.0 );
MCMoveRelative( hCtlr, 1, 122.5 );
```

Constant Velocity Motion

To move a servo at a continuous velocity until commanded to stop:

```
// Enable the axis
// Enable Velocity mode
// define maximum velocity
// define acceleration/deceleration
// define the direction (positive or negative) of the move
// begin motion of axis 1
// wait for digital I/O #4 to be true
// reduce velocity
// wait for digital I/O #2 to be true
// stop the motion of axis 1


MCEnableAxis( hCtlr, 1, TRUE );
MCSetOperatingMode( hCtlr, 1, 0, MC_MODE_VELOCITY );
MCSetVelocity( hCtlr, 1, 10000.0 );
MCSetAcceleration( hCtlr, 1, 100000.0 );
MCSetDirection( hCtlr, 1, POSITIVE );
MCGo( hCtlr, 1 );
MCWait For DigitalIO( hCtlr, 4, TRUE );
MCSetVelocity( hCtlr, 1, 5000.0 );
MCGo( hCtlr, 1 );
MCWait For DigitalIO( hCtlr, 2, TRUE );
MCStop( hCtlr, 1 );
```



Jogging

In some applications it may be necessary to have a means of manually positioning the motors. Since the DCX is able to control the motion of servos with precision at both low and high speeds, all that is required to support manual positioning is: .

- A PC with a game port
- A PC joystick
- PC based software that positions the axes in Velocity mode

Jogging without writing software
One of the tools provided with the MCAPI is the Joystick Demo. This tool allows the user to configure and then jog one or two axes.

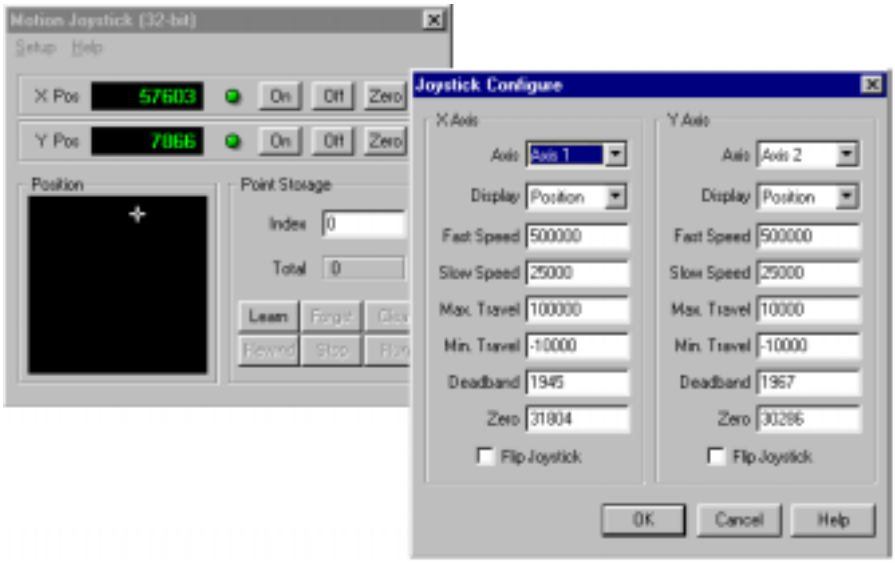
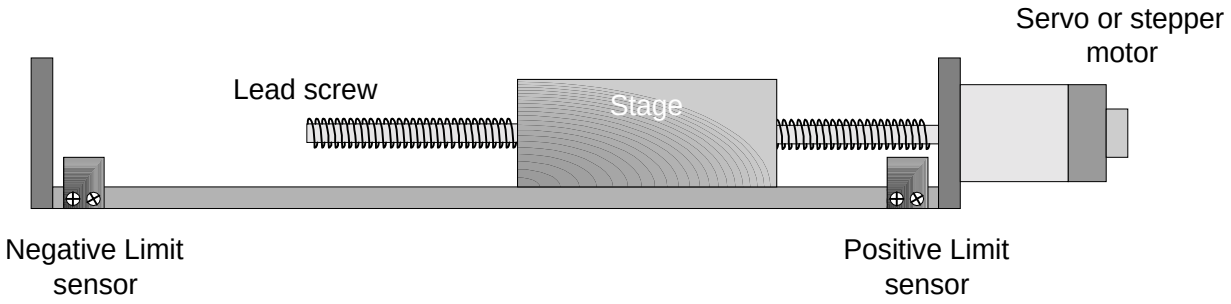


Figure 26: Joystick Demo program

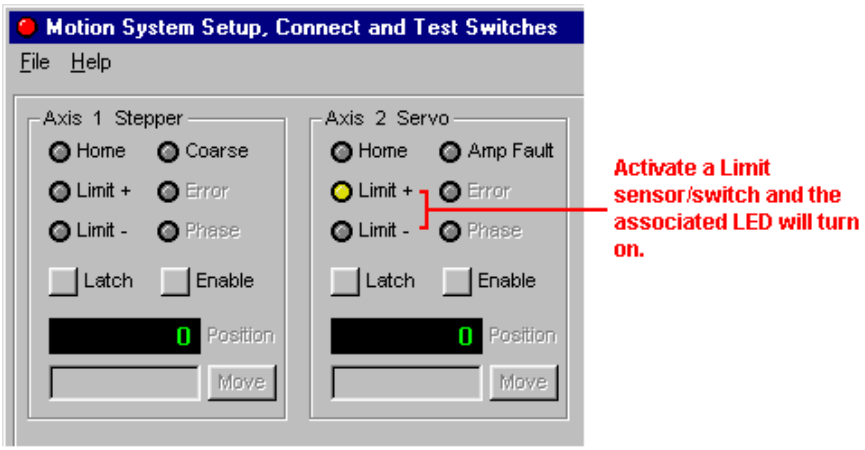
Using the Joystick Demo in your application program
After the MCAPI has been installed the source files for the Joystick Demo are available in the Motion Control folder \Program Files\Motion Control\Motion Control API\Sources\Joy.

Defining Motion Limits

The DCX-PCI100 supports both ‘hard coded’ handling of End of travel or 'Hard' limit switch/sensors and programmable soft limits.



Hard Limits
The Limit + /- inputs of all MC1XX motion control modules default to TTL low true operation. When a limit input signal is pulled low (> 0.7V), the DCX will indicate that the input is active. Use the Motion Integrator Motion System Setup Test Panel to test the limit sensors, wiring, and MC100, MC110/110 operation.



When limit error checking is enabled by the ***MCSetLimits( )*** function, the limit tripped flags (MC_STAT_PLIM_TRIP and MC_STAT_MLIM_TRIP) indicate an error condition. For a normally closed limit switch, the MC_LIMIT_INVERT parameter must be used to re-define the active level

of the limit circuit.

The limit LED's of the Motion Integrator Test Panel display the current state (MC_STAT_PLIM and MC_STAT_MLIM), not the 'tripped' flag (MC_STAT_PLIM_TRIP and MC_STAT_MLIM_TRIP) of the limit inputs. The Motion Integrator Test Panel will indicate that a normally closed limit switch is active until the switch is opened.

The DCX supports two levels of limit switch handling:

Auto axis disable
Simple monitoring

The MCAPI function ***MCSetLimits( )*** allows the user to enable the Auto Axis Disable capability of the DCX. This feature implements a hard coded operation that will stop motion of an axis when a limit switch is active. This background operation requires no additional DCX processor time, and once enabled, requires no intervention from the user's application program. However it is recommended that the user periodically check for a limit tripped error condition using the ***MCGetStatus( )***, ***MCDecodeStatus( )*** functions. The ***MCSetLimit( )*** function provides the following limit flags:

Flag	***Description***
MC_LIMIT_PLUS	Enables the Positive/High hard limit
MC_LIMIT_MINUS	Enables the Negative/Low hard limit
MC_LIMIT_BOTH	Enables the Positive and Negative hard limits
MC_LIMIT_OFF	Turn off the axis when the hard limit input 'goes' active
MC_LIMIT_ABRUPT	Stop the axis abruptly when the hard limit input goes active
MC_LIMIT_SMOOTH	Decelerate and stop the axis when the hard limit input goes active
MC_LIMIT_INVERT	Invert the active level of the hard limit input to high true. Typically used for normally closed limit sensors

When a limit event occurs, motion of that axis will stop and the error flags (MC_STAT_ERROR and MC_STAT_PLIM_TRIP or MC_STAT_MLIM_TRIP) will remain set until the motor is turned back on by ***MCEnable( )***. The axis must then be moved out of the limit region with a move command (**MCMoveAbsolute( )**, ***MCMoveRelative( )***).

```
// Set the both hard limits of axis 1 to stop smoothly when tripped, ignore
// soft limits:
//

MCSetLimits( hCtlr, 1, MC_LIMIT_BOTH | MC_LIMIT_SMOOTH, 0, 0.0, 0.0 );



// Set the positive hard limit of axis 2 to stop by turning the motor off.
// Because axis 2 uses normally closed limit switches we must also invert the
// polarity of the limit switch. Soft limits are ignored.

MCSetLimits( hCtlr, 2, MC_LIMIT_PLUS | MC_LIMIT_OFF | MC_LIMIT_INVERT, 0, 0.0,
0.0 );
```

If the user does not want to use the Auto Axis Disable feature, the current state of the limit inputs can be determined by polling the DCX using the ***MCGetStatus( )***, ***MCDecodeStatus( )*** functions. The flag

for testing the state of the Limit + input is **MC_STAT_INP_PLIM**. The flag for testing the state of the Limit - input is **MC_STAT_INP_MLIM**.

Soft Limits

Soft motion limits allow the user to define an area of travel that will cause a DCX error condition. When enabled, if an axis is commanded to move to a position that is outside the range of motion defined by the ***MCSetLimit( )*** function, an error condition is indicated and the axis will stop. The ***MCSetLimit( )*** function provides the following limit flags:

Flag	***Description***
MC_LIMIT_PLUS	Enables the High/Positive soft limit
MC_LIMIT_MINUS	Enables the Low/Negative soft limit
MC_LIMIT_BOTH	Enables the High and Low soft limits
MC_LIMIT_OFF	Turn off the axis when the hard limit input 'goes' active
MC_LIMIT_ABRUPT	Stop the axis abruptly when the hard limit input goes active
MC_LIMIT_SMOOTH	Decelerate and stop the axis when the hard limit input goes active

When a soft limit error event occurs, the error flags (**MC_STAT_ERROR** and **MC_STAT_PSOFT_TRIP** or **MC_STAT_MSOFT_TRIP**) will remain set until the motor is turned back on by ***MCEnable( )***. The axis must then be moved back into the allowable motion region with a move command (**MCMoveAbsolute( )**, ***MCMoveRelative( )***).

```
// Assume axis 3 is a linear motion with 500 units of travel. Set the both
// hard limits of this axis to stop abruptly. Set up soft limits that will
// stop the motor smoothly 10 units from the end of travel (i.e. at 10
// and 490).

MCSetLimits( hCtlr, 3, MC_LIMIT_BOTH | MC_LIMIT_ABRUPT, MC_LIMIT_BOTH |
MC_LIMIT_SMOOTH, 10.0, 490.0 );
```

Homing Axes

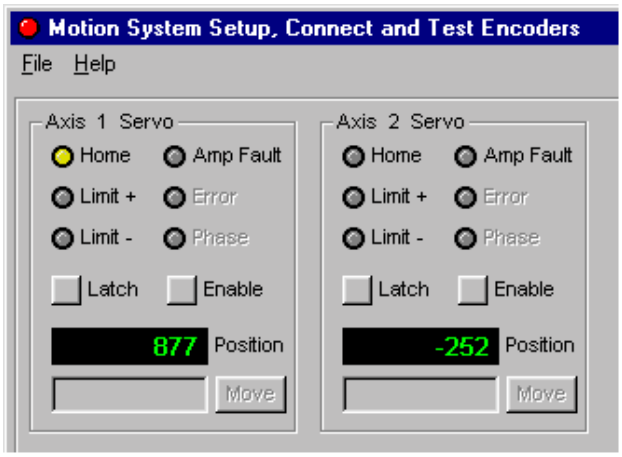
When power is applied or the DCX is reset, the current position of all servo axes are initialized to zero. If they are subsequently moved, the controller will report their positions relative to the position where they were last initialized. At any time the user can call the **MCSetPosition( )** function to re-define the position of an axis.

In most applications, there is some position/angle of the axis (or mechanical apparatus) that is considered 'home'. Typical automated systems utilize electro-mechanical devices (switches and sensors) to signal the controller when an axis has reached this position. The controller will then define the current position of the axis to a value specified by the user. This procedure is called a homing sequence. The DCX is not shipped from the factory programmed to perform a specific homing operation. Instead, it has been designed to allow the user to define a custom homing sequence that is specific to the system requirements. The DCX provides the user with two different options for homing axes:

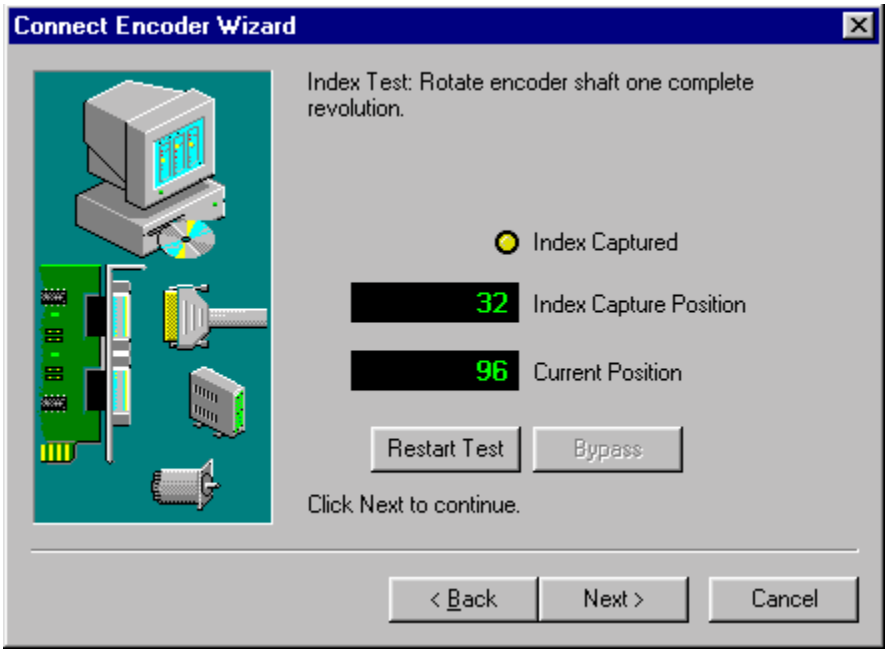
1) **High level function calls using the MCAPI** - Easy to program homing sequences using MCAPI function calls.

2) **MCCL Homing macro's stored in on-board, non-volatile FLASH memory** - When executed as background tasks, MCCL homing macro's allow the user to home multiple axes simultaneously.

Verifying the operation of the Home Sensor
Most motion applications will utilize a home sensor as a part of the homing sequence. Use Motion Integrator's Connect Axis I/O Wizard or Motion System Setup Test Panel to verify the proper operation of the encoder index.



Verifying the operation of the Index Mark of an Encoder
Most servo applications will utilize the Index mark of the encoder to define the 'home' position of an axis. Use Motion Integrator's Connect Encoder Wizard to verify the proper operation of the encoder index.



Homing a Rotary Stage (servo) with the Encoder Index

Many servo motor encoders generate an index pulse once per rotation. For a multi turn rotary stage, where one rotation of the encoder equals one rotation of the stage, an index mark alone is sufficient for homing the axis. When an axis need only be homed within 360 degrees no additional qualifying sensors (coarse home) are required. The following MCAPI and MCCL command sequences will home a multi turn rotary stage:

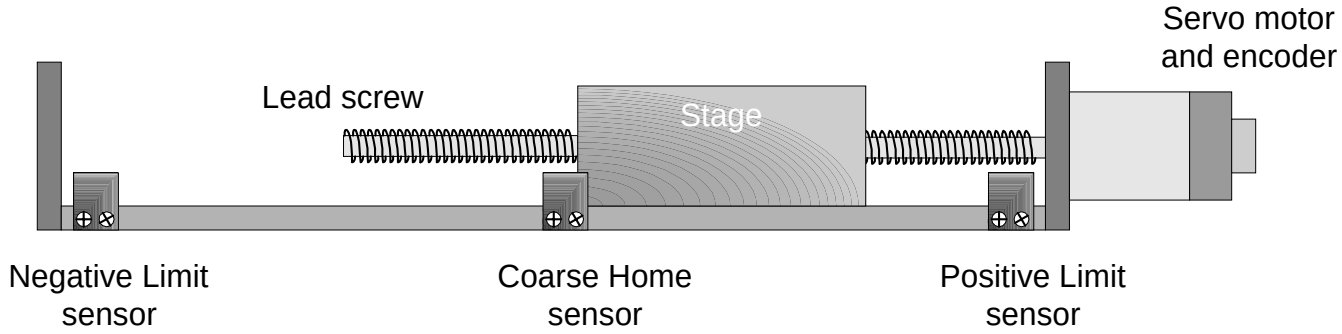
```
// MCAPI rotary axis homing sequence
//
// Configure axis, start homing
//
MCSetOperatingMode( hCtlr, 1, 0, MC_MODE_VELOCITY );
MCDirection( hCtlr, 1, MC_DIR_POSITIVE );
MCSetVelocity( hCtlr, 1, 5000.0 );
MCGo( hCtlr, 3 );

// Stop when index mark captured
//
MCFindIndex( hCtlr, 1, 0.0 );
MCStop( hCtlr, 1 );
MCWaitForStop( hCtlr, 1, 0.01 );

// Move back to location of index mark
//
MCSetOperatingMode( hCtlr, 1, 0, MC_MODE_POSITION );
MCEnableAxis( hCtlr, 1, TRUE );
MCMoveAbsolute( hCtlr, 1, 0.0 );
MCWaitForStop( hCtlr, 1, 0.01 );


;MCCL homing sequence executed as a background task
;
GT0,1VM,1DI0,1SV50000,1GO,1FI0,1ST,1WS.01,1PM,1MN,1MA0,1WS.01
```

Homing a Servo Axis with Coarse Home and Encoder Index Inputs

A typical axis will incur multiple rotations of the motor/encoder over the full range of travel. This type of system will typically utilize a coarse home sensor to qualify which of the index pulses is to be used to home the axis. The Limit Switches (end of travel) provide a dual purpose:

1) Protect against damage of the mechanical components.
2) Provide a reference point during the initial move of the homing sequence

The following diagram depicts a typical linear stage.



When power is applied or the DCX is reset, the position of the stage is unknown. The following MCAPI and MCCL homing samples will move the stage in the positive direction. If the coarse home sensor ‘goes active’ before the positive limit sensor, the Find Index command will redefine the position of the axis when the index mark is captured. If the positive limit sensor ‘goes active’, the stage will change direction, until **both** the coarse home sensor **and** the encoder index are active, at which point the position will be redefined.

```
// MCAPI homing sequence (using positive limit, coarse home, and
// index mark)
//
// Enable limit switches, start velocity mode move
//
MCSetLimits( hCtlr, 1, MC_LIMIT_SMOOTH | MC_LIMIT_HIGH | MC_LIMIT_LOW, 0, 0, 0
);
MCSetOperatingMode( hCtlr, 1, 0, MC_MODE_VELOCITY );
MCSetVelocity( hCtlr, 1, 10000.0 );
MCDirection( hCtlr, 1, MC_DIR_POSITIVE );
MCGoEx( hCtlr, 1, 0.0 ) );

//
// Wait for coarse home or positive limit inputs
dwStatus = MCGetStatus( hCtlr, 1);
while  (! MCDecodeStatus( hCtlr, dwStatus, MC_STAT_INP_HOME) ||
       ! MCDecodeStatus( hCtlr, dwStatus, MC_STAT_PLIM_TRIP)) {
       dwStatus = MCGetStatus( hCtlr, 1);
}
```

```
// If positive limit switch active
//
dwStatus = MCGetStatus( hCtlr, 1);
if (! MCDecodeStatus( hCtlr, dwStatus, MC_STAT_PLIM_TRIP)) {
   MCEnableAxis( hCtlr, 1, TRUE );
   MCDirection( hCtlr, 1, MC_DIR_NEGATIVE );
   MCSetVelocity( hCtlr, 1, 10000.0 );
   MCGoEx( hCtlr, 1, 0.0 ) );
   MCWaitForEdge( hCtlr, 1, TRUE );
   MCStop( hCtlr, 1 );
   MCWaitForStop( hCtlr, 1, 0.1 );
}

//  Once within Coarse Home sensor range, reduce velocity
//  Move until Coarse Home sensor is no longer active
//
MCDirection( hCtlr, 1, MC_DIR_NEGATIVE );
MCSetVelocity( hCtlr, 1, 2000.0 );
MCGoEx( hCtlr, 1, 0.0 ) );
MCWaitForEdge( hCtlr, 1, FALSE );
MCStop( hCtlr, 1 );
MCWaitForStop( hCtlr, 1, 0.1 )

// When Coarse Home no longer is active, reduce velocity
// Move back towards until index mark is captured
//
MCDirection( hCtlr, 1, MC_DIR_POSITIVE );
MCSetVelocity( hCtlr, 1, 1000.0 );
MCGoEx( hCtlr, 1, 0.0 ) );
MCWaitForEdge( hCtlr, 1, TRUE );
MCFindIndex( hCtlr, 1, 0.0 );
MCStop( hCtlr, 1 );
MCWaitForStop( hCtlr, 1, 0.1 )

// Issue position mode move to location of index mark (position 0)
//
MCSetOperatingMode( hCtlr, 1, 0, MC_MODE_POSITION );
MCEnableAxis( hCtlr, 1, TRUE );
MCMoveAbsolute( hCtlr, 1, 0.0 );
MCWaitForStop( hCtlr, 1, 0.1 );
```

```
; MCCL homing sequence (using positive limit, coarse home, and index mark)

MD1,1LM2,1LN3,MJ10                      ;enable limits, call homing macro
MD10,1VM,1SV10000,1DI0,1GO,LU”STATUS”,1RL@0,IS25,MJ11,NO,IS17,MJ12,NO,JR-8
                                        ;start move, test for sensors (home
                                        ;and +limit)
MD11,1ST,1WS.01,1DI1,1GO,1WE1,1ST,1WS.1,1DI0,1GO,1WE0,1FI0,1ST,1WS.01,1PM,1MN,
1MA0
                                        ;if home sensor true, initialize on
                                        ;index pulse
MD12,1WS0.01,1MN,1DI1,1GO,1WE0,MJ11     ;move negative until home true
```

An axis can be homed even if no index mark or coarse home sensor is available. This method of homing utilizes one of the limit (end of travel) sensors to also serve as a home reference. Please note that this method

is not recommended for applications that require **high repeatability and accuracy**. To achieve the highest possible accuracy when using this method, significantly reduce the velocity of the axis while polling for the active state of the limit input.

The following MCAPI and MCCL sequences will home an axis at the position where the positive limit sensor 'goes active':

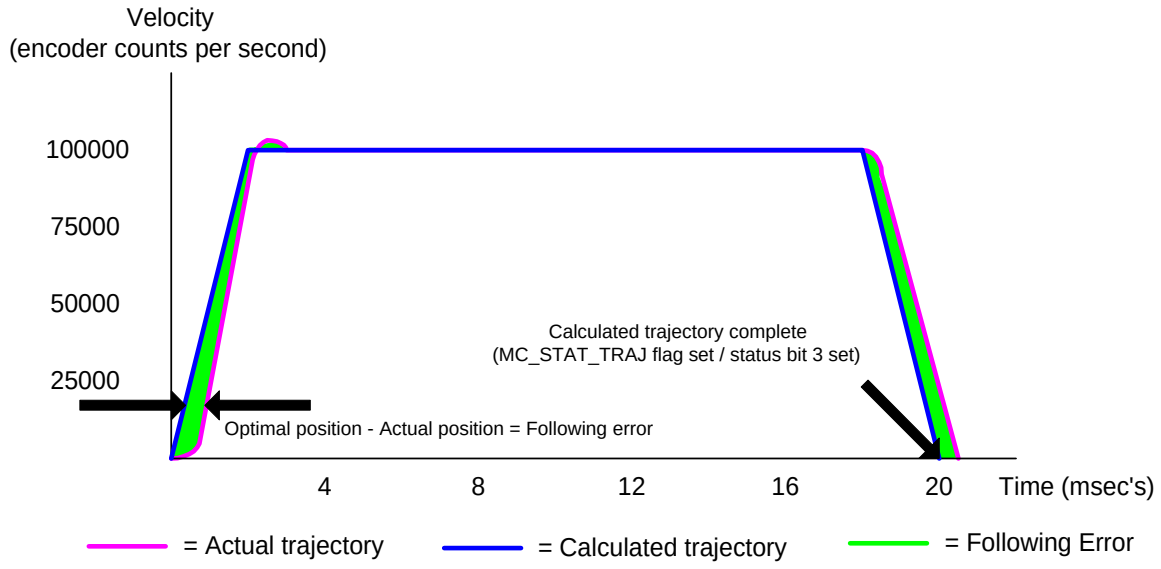
```
// MCAPI homing sequence (using positive limit index mark)
//
// Enable limit switches, start velocity mode move
//
MCSetLimits( hCtlr, 1, MC_LIMIT_SMOOTH | MC_LIMIT_HIGH | MC_LIMIT_LOW, 0, 0, 0
);
MCSetOperatingMode( hCtlr, 1, 0, MC_MODE_VELOCITY );
MCSetVelocity( hCtlr, 1, 10000.0 );
MCDirection( hCtlr, 1, MC_DIR_POSITIVE );
MCGoEx( hCtlr, 1, 0.0 ) );

//
// Wait for positive limit inputs
dwStatus = MCGetStatus( hCtlr, 1);
while  (! MCDecodeStatus( hCtlr, dwStatus, MC_STAT_PLIM_TRIP)) {
       dwStatus = MCGetStatus( hCtlr, 1);
}


// Once the positive limit switch is active, move negative until switch is inactive
//
MCEnableAxis( hCtlr, 1, TRUE );
MCDirection( hCtlr, 1, MC_DIR_NEGATIVE );
MCSetVelocity( hCtlr, 1, 1000.0 );
MCGoEx( hCtlr, 1, 0.0 ) );
dwStatus = MCGetStatus( hCtlr, 1);
if (! MCDecodeStatus( hCtlr, dwStatus, MC_STAT_INP_PLIM)) {
   dwStatus = MCGetStatus( hCtlr, 1)
}

// Stop the axis and define the leading edge of the limit switch as position 0
//
MCAbort( hCtlr, 1 );
MCWaitForStop( hCtlr, 1, 0.1 );
MCSetPosition( hCtlr, 1, 0.0 );
MCSetOperatingMode( hCtlr, 1, 0, MC_MODE_POSITION );
MCEnableAxis( hCtlr, 1, TRUE );
MCMoveAbsolute( hCtlr, 1, -100.0 );



; MCCL homing sequence (using positive limit, coarse home, and index mark)

MD1,1LM2,1LN3,MJ10                        ;call homing macro
MD10,1VM,1DI0,1GO,LU"STATUS",1RL@0,IS17,MJ11,NO,JR-5
                                          ;move and poll the Limit + sensor
MD11,1WS0.01,1MN,1DI1,1SV1000,1GO,LU"STATUS",1RL@0,IC28,MJ12,NO,JR-5
                                          ;move negative until limit + inactive
MD12,1AB,1WS.1,1DH0,1PM,1MN,1MA-100       ;stop when limit + not active, define
                                          ;position as 0. Move to position –100.
```

Motion Complete Indicators

When the DCX motion controller receives a move command, the DCX-PCI100 motherboard sends a new target position to the appropriate servo control module (DCX-MC100, MC110 or DCX-MC110). The servo module then calculates a trapezoidal velocity profile based on the:

- New target position
- Current settings for maximum velocity and acceleration/deceleration.

The trapezoidal velocity profile calculations result in position points that are evenly separated in time (by 341 usec's, the period of the PID filter). These calculated position points are known as Optimal Positions. During a servo axis move there will always be some difference between the calculated position (Optimal Position) and the current position, this difference is known as the Following Error.



As the end of a move approaches, once the optimal position of an axis is equal to the move target, the 'digital trajectory' of the move has been completed and the **MC_STAT_TRAJ** status flag (MCCL status trajectory complete bit 3) will be set. As shown in the preceding diagram, if a following error is present during a move the axis will continue to move after the trajectory is complete, until the following error is minimized.

This status flag is the conditional component of the ***MCIsStopped( )*** and ***MCWaitForStop( )*** functions. As shown above, a following error can cause MC_STAT_TRAJ to be set **before the axis has reached its target**. Issuing **MCIsStopped( )** with a timeout value specified or **MCWaitForStop( )** with a *Dwell* time specified allows the user to delay execution move has been completed (following error = 0). In the example below, the **MCWaitForStop( )** command includes a Dwell of 5 msec's, allowing the axis to stop and settle.

```
MCMoveRelative( hCtlr, 2, 500.0 );      // move 500 counts
MCWaitForStop( hCtlr, 2, 0.005 );       // wait till MC_STAT_TRAJ set plus
                                        // 5 msec's
```

Another method of indicating the end of a move is to use ***MCIsAtTarget( )*** or ***MCWaitForTarget( )*** . To satisfy the conditions of ***MCIsAtTarget( )*** and ***MCWaitForTarget( )*** , the axis must be within the **Dead band** range for the time specified by **DeadbandDelay**, both of which are defined within the **MCMotion** data structure.

The Dead band and DeadbandDelay are used to define an acceptable 'at target range' for the axis. The Dead band defines an 'at target' range (in encoder counts) of an axis. The DeadbandDelay defines the amount of time that the axis must remain within the 'at target' range before the status flag MC_STAT_AT_TARGET bit will be set.

```
MCMoveRelative( hCtlr, 1, 1250.0 );     // move 1250 counts
MCWaitForTarget( hCtlr, 1, 0.005 );     // wait till MC_STAT_TRAJ set plus
                                        // msec's
```

On the Fly changes

During a point to point or constant velocity move of one or more axes, the DCX supports 'on the fly' changes of:

- Target
- Maximum Velocity
- Servo PID parameters

Changes made to any or all of these motion settings while an axis is moving will take affect within 8 msec's.

If an "on the fly" target position change requires a change of direction the axis will first decelerate to a stop. The axis will then move in the opposite direction to the new target. This will occur if:

1) The new target position is in the opposite direction of the current move
2) A **'near target**' is defined. A near target is a condition where the current deceleration rate will not allow the axis to stop at the new target position. In this case the axis will decelerate to a stop at the user define rate, which will result in an overshoot. The axis will then move in the opposite direction to the new target.

If an on the fly change requires the axis to change direction, the DCX command interpreter will stall, not accepting any additional commands, until the change of direction has occurred (deceleration complete).

The DCX-PCI100 does not support changing the acceleration on the fly

Note – Changing the PID parameters (Proportional gain, Derivative gain, Integral gain) ‘on the fly’ may cause the axis to jump, oscillate, or ‘error out’.

‘On the fly’ velocity changes will not take effect until after the axis has be re-enabled (***MCEnableAxis*** function or ***a*GO** command).

Save and Restore Axis Configuration

The MCAPI Motion Dialog library includes ***MCDLG_SaveAxis( )*** and ***MCDLG_RestoreAxis( )***. These high level dialogs allow the programmer to easily maintain and update the settings for servo axes.

MCDLG_SaveAxis( ) encodes the motion controller type and module type into a signature that is saved with the axis settings. ***MCDLG_RestoreAxis( )*** checks for a valid signature before restoring the axis settings. If you make changes to your hardware configuration (i.e. change module types or controller type) ***MCDLG_RestoreAxis( )*** will refuse to restore those settings.

You may specify the constant **MC_ALL_AXES** for the *wAxis* parameter in order to save the parameters for all axes installed on a motion controller with a single call to this function.

If a NULL pointer or a pointer to a zero length string is passed as the *PrivateIniFile* argument the default file (MCAPI.INI) will be used. Most applications should use the default file so that configuration data may be easily shared among applications. Acceptance of a pointer to a zero length string was included to support programming languages that have difficulty with NULL pointers (e.g. Visual Basic).

Chapter Contents

Chapter

8

Application Solutions

Converting from an ISA bus DCX-PC100 motion controller

DCX-PCI100 Enhancements
The DCX-PCI100 motion control motherboard was designed specifically to provide PCI bus support for DCX-MC100 and DCX-MC110 users. With the added processing power of the MIPS CPU the following enhancements are now available to MC100/MC110 users:

- Faster command execution - typical execution time decreases from 750 usec's to 50 usec's
- User unit scaling for distance, rate, time, and position offsets
- Multi-tasking for MCCL subroutines
- Firmware stored in on board FLASH for easy firmware upgrades by the user
- Additional axis status data (Hard & Soft Limit Mode, user scaling settings, etc…)
- Invert Limit option supports both normally open and normally closed end of travel sensors
- User defined motion limits (soft limits)
- Graphical Servo Tuning program
- Variables for reading axis data (position, status, velocity, etc…)
- Floating point and integer parameters
- Additional error reporting

Required changes when converting to DCX-PCI100
The DCX-PCI100 enhancements precluded 100% backward compatibility with ISA applications. For ISA-based (DCX-PC100) applications, programmed using either the MCAPI function library or MCCL commands, when migrating to PCI-based DCX-PCI100, the following changes will be required:

- The DCX-PCI100 **must be installed in a computer running Windows** 2000/NT/ME/98, it does not support Windows 3.X or 95.
- The PCI bus was not designed to carry high current DC voltages to PCI bus cards. To provide the necessary current for DCX-MC110 Direct Motor Drive modules (as much as 4.0 amps) the

DCX-PCI100 Motion Control Motherboard includes an auxiliary motor power connector (J33). The pinout of connector J33 matches the power supply connections for 5 ¼ " floppy disk drives and HDD's (Hard Disk Drive). A Floppy Drive Power Cable Splitter is used to directly connect the PC's +12 VDC supply to the DCX-PCI100. Floppy Drive Power Cable Splitters are available at most computer and electronic supply stores, or can be purchased directly from PMC (P/N 71.060.A).

- The DCX-PCI100 does not support DOS application programming, but it does support 32-bit Console Mode applications. For additional information please refer to **TechNOTE 1013** "**Porting Legacy MS-DOS Motion Applications to Windows NT**".
- Upgrade the MCAPI – the DCX-PCI100 requires MCAPI revision 3.1.00 or higher. For additional information on installing the MCAPI (and removing older revisions of the MCAPI) please refer to the **DCX-PCI100 User Manual, chapter 2, Controller and Software Installation**.
- Trajectory parameters (Set Velocity, Set Acceleration) are expressed in encoder counts per second (velocity = counts/sec, accel/decel = counts/sec/sec) instead of encoder counts per sample period (velocity = counts * .000341 *65,536; accel/decel = counts *.000341*.000341 * 65,536)
- Time units (WAit, Wait for Stop) are expressed in seconds instead of milliseconds (1WS5 converts to 1WS0.005)
- The Motor Table no longer uses hard coded addressing. For example, the command 1RL0 would load the status word of axis #1 into the accumulator of an ISA based DCX-PCI100. For PCI based applications, the user first issues the Look Up variable command with the parameter equal to the variable name (enclosed in quotation marks). Then issue a read command (long, word, double, etc…) to the appropriate axis:

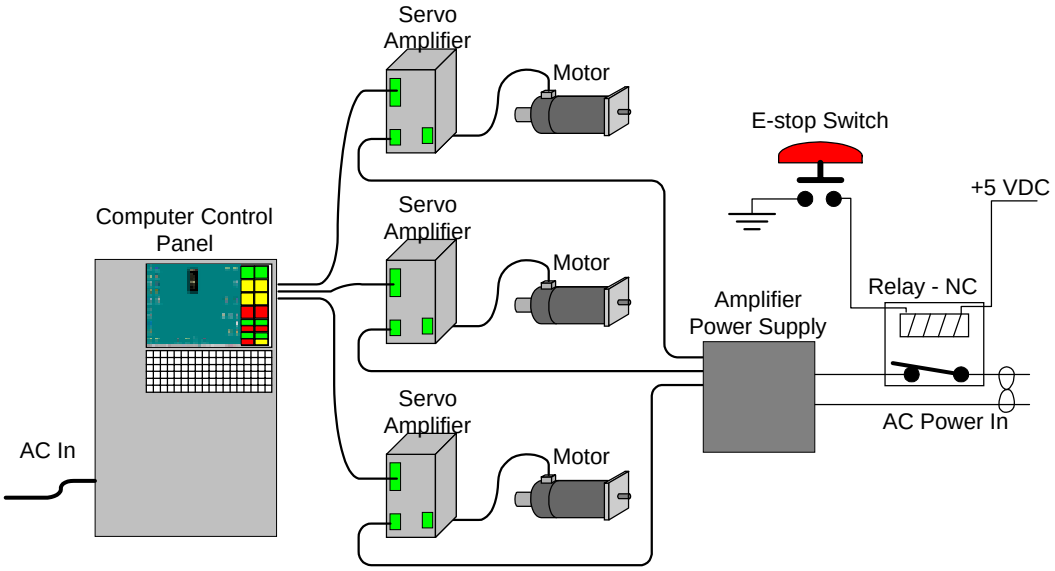
```
LU"STATUS",1RL@0                ;load axis #1 status into accumulator
```

Features no longer supported

- Manual positioning (jogging) by activating the Jog Right and Jog Left inputs
- Motherboard based general purpose I/O. The DCX-PC100 has 16 general purpose digital I/O and 4 eight bit analog inputs. The DCX-PCI100 motherboard does not provide any general purpose I/O. The DCX-MC400 Digital I/O module and the DCX-MC500 Analog I/O module are supported by the DCX-PCI100, allowing the user to add I/O capability.
- The DCX-PCI100 does not support RS-232 or IEEE-488 communication interfaces

Emergency Stop

Many applications that use motion control systems must accommodate regulatory requirements for immediate shut down due to emergency situations. Typically these requirements do not allow an emergency shut down to be controlled by a programmable computing device. The drawing below depicts an application where an emergency stop must be a completely 'hard wired' event.

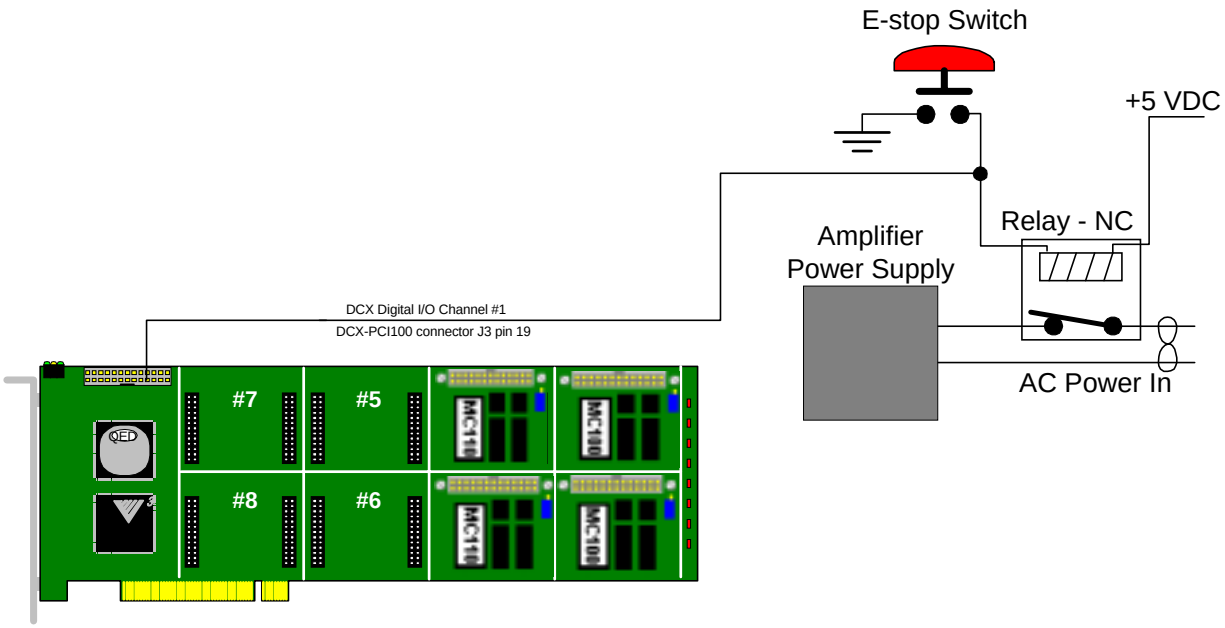


This 'hard wired' E-stop circuit uses a relay to disconnect power from the servo amplifiers. The motors and amplifiers would certainly be disabled, but the motion controller and the application program will have no indication that an error condition exists.

Wiring the E-Stop switch to the DCX
There are two ways to wire the DCX so that it can monitor the E-stop switch:

1) Connect the E-stop switch to one of the general purpose digital I/O lines
2) Connect all of the Amplifier Fault inputs to the E-stop switch

E-stop switch connected to DCX General Purpose Digital Input
Wire the E-stop switch to a general purpose digital I/O (channel #1). Each DCX digital channel has a 4.7K resistor pulled up to +5 volts. A background task is used to monitor the state of the input. If the channel is configured for low 'low true' operation, the input (from the E-stop switch) will report its state as 'off' until the E-stop switch is activated. The ***WaitForDigitalIO*** function will stay active in background until the input 'goes true'.



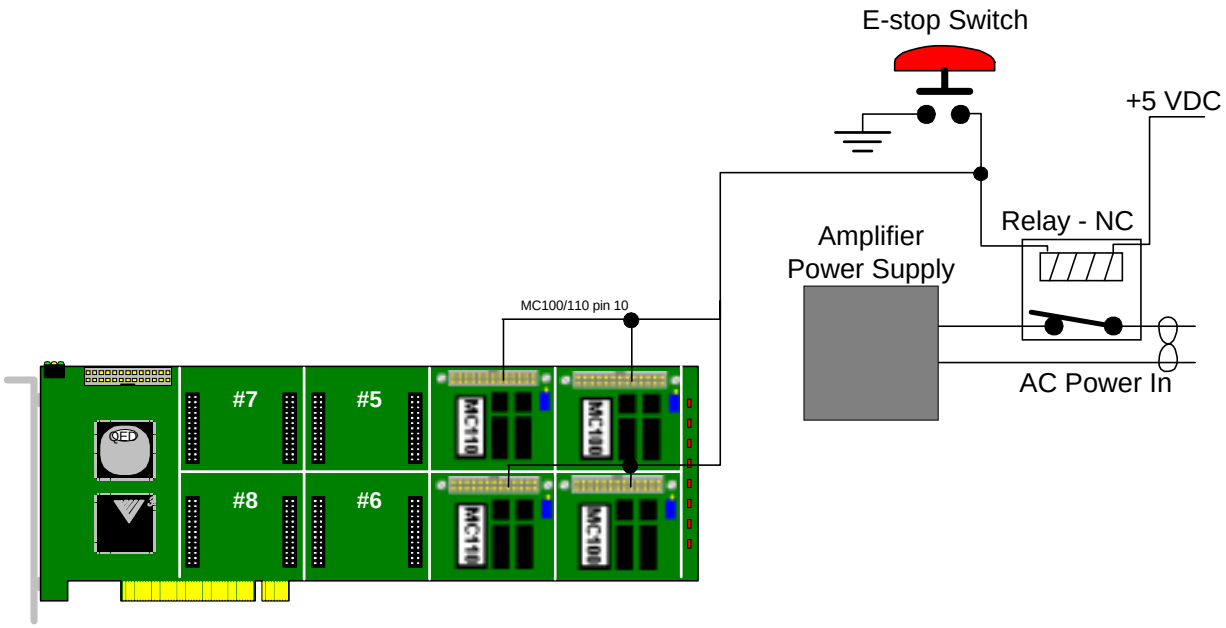

```
if (MCBlockBegin ( hCtlr,MC_BLOCK_TASK, 0) ==MCERR_NOERROR )    {
    MCSetRegister ( hCtlr, 100, 0, MC_TYPE_LONG);
    MCConfigureDigitalIO ( hCtlr, 1, MC_DIO_LOW );
    MCWaitForDigitalIO ( hCtlr, 1, TRUE );
    MCSetRegister hCtlr, 100, 1, MC_TYPE_LONG);
    MCEnableAxes( hCtlr, MC_ALL_AXES, FALSE );
    MCBlockEnd ( hCtlr, NULL);
}


// periodically poll the user register #100 for a value of 1. If true the user
// can jump to an E-stop handling routine.

MCGetUserRegister (hCtlr, 100, &Estop, MC_TYPE_LONG);
```

E-stop switch connected to Amplifier Fault servo module input

The Amplifier Fault input of MC200 and MC210 servo modules can be used to disable motion with no user software action required. The E-stop switch is wired to the Amplifier Fault input (connector J3 pin 10) of **each servo module**. Auto shut down of motion upon activation of the Amplifier Fault input is enabled by the **MCMotion** structure member **EnableAmpFault**. When the E-stop switch is activated:

1) The axis is disabled (PID loop terminated, Amplifier Enable output turned off)
2) The status flag **MC_STAT_AMP_FAULT** will be set for each axis
3) The status flag **MC_STAT_ERROR** will be set for each axis

When the E-stop condition has been cleared, motion can be resumed after issuing the ***MCEnableAxis*** function with the parameter *wAxis* set to **MC_ALL_AXES**.



Encoder Rollover

The DCX motion controller provides 30 bit position resolution, resulting in a position range of –1,073,741,823 to 1,073,741,823. For an application where the axis is moving at maximum velocity (750 thousand encoder counts per second), the encoder would rollover in approximately 23 minutes. When the encoder rolls over, the reported position of the axis will change from a positive to a negative value. For example, if the axis is at position 2,147,483,647 the next positive encoder count will cause the DCX to report the position as –2,147,483,647.

If a user scaling other than 1:1 has been defined the DCX controller will report the position in user units. The reported position at which the value will rollover is based on the user scaling. If user scaling is set to 10,000 encoder counts to one position unit, the reported position will rollover at position 214,748.3647. The next positive encoder count will cause the DCX to report the position as –214,748.3647.

Encoder rollover during Position Mode moves
The DCX will not accept a Position Mode move that exceeds the rollover point, this would essentially be handled as an error condition, except the PID filter will remain enabled.

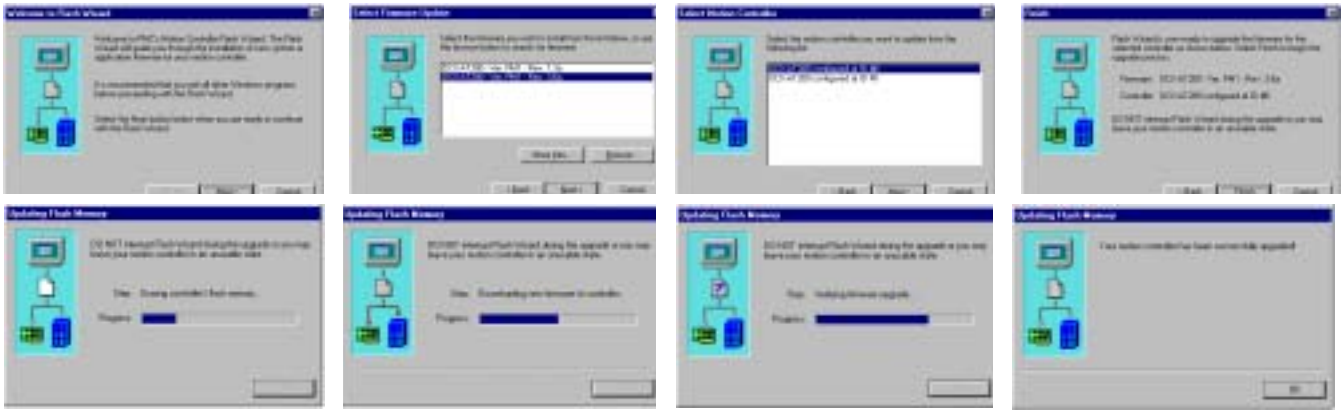
Encoder rollover during Velocity Mode moves
No disruption or unexpected motion will occur if a rollover occurs during a Velocity mode (***MCSetOperatingMode***, **MC_MODE_VELOCITY**) move. However, once the rollover point has been crossed, the position reported by the ***MCTellPosition*** function will longer be valid.

Prior to executing a velocity mode move in which the encoder position may rollover the axis **must** be homed (MCFindIndex or MCSetPosition) to position 0. Defining a offset to the home position will cause the axis to pause at the rollover point.

Flash Memory Firmware Upgrade

Each time the PC is re-booted (reset or power cycle) the operating code (typically called firmware) for the DCX-PCI100 is loaded into on-board SDRAM (Static Dynamic Random Access Memory). The source files for the operating code is written to the PC's hard disk drive during the installation of the MCAPI.

PMC's **Flash Wizard** (the DCX-PCI100 requires Flash Wizard rev. 2.20 or higher) is a windows utility that allows the user to easily update the operational code. Code updates are available from the **MotionCD** or from PMC's web site **www.pmccorp.com**.



With Windows 98 and MCAPI 3.1.000 a verification error may will occur during code download. To complete the firmware upgrade close Flash Wizard and restart the PC.

Learning/Teaching Points

As many as 256 points can be stored for ***each axis*** in the DCX's point memory by using the **MCLearnPoint( )** function. A stored point can be either the actual position of an axis (**MC_LRN_POSITION**) or the target position of an axis (**MC_LRN_TARGET**).

The value **MC_LRN_POINT** would typically be used in conjunction with jogging. The operator would jog the axes along the desired path, issuing the ***MCLearnPoint( )*** command at regular intervals. The ***MCMovePoint( )*** command would then be used to 'play back' the path traversed by the operator.

For applications where the target point data was previously recorded and stored in the PC, the value **MC_LRN_TARGET** would be used to load the target points into the DCX.

Once all points have been stored, the axes are commanded to move to the stored positions with ***MCMoveToPosition( )***. The parameter *wIndex* indicates to which stored point the axis should move.

```
// Move axis 1 and store position in consecutive point storage locations.

WORD wIndex;
MCEnableAxis( hCtlr, 1, TRUE );         // motor on
MCGoHome( hCtlr, 1 );                   // start from absolute zero
MCWaitForStop( hCtlr, 1, 0.100 );

for (wIndex = 0; wIndex < 5; wIndex++) {
   MCMoveRelative( hCtlr, 1, 1234.0 );  // move
   MCWaitForStop( hCtlr, 1, 0.100 );    // are we there yet?
   MCLearnPoint( hCtlr, 1, wIndex, MC_LRN_POSITION );
}


// Store several positions for axis 4 without actually moving the axis. Note // that
axis is disabled with MCEnableAxis( ) prior to storing positions

WORD wIndex;
MCEnableAxis( hCtlr, 4, FALSE );        // motor off
for (wIndex = 0; wIndex < 5; wIndex++) {
   MCMoveRelative( hCtlr, 4, 2468.0 );  // nothing actually moves
   MCLearnTarget( hCtlr, 4, wIndex, MC_LRN_TARGET );
}


// This example moves to the stored positions, dwelling for 0.2 seconds at
// each point.

WORD wIndex;
MCEnableAxis( hCtlr, 4 );               // enable axis
for (wIndex = 0; wIndex < 5; wIndex++) {
   MCMoveToPoint( hCtlr, 4, wIndex );   // move to next point
   MCWaitForStopped( hCtlr, 4, 0.2 );
}
```

Record Motion Data

The DCX supports capturing and retrieving motion data from servo axes (MC100, MC110). Captured position data is typically used to analyze servo motor performance and PID loop tuning parameters. The MCAPI function ***MCCaptureData( )*** is used to acquire motion data for a servo axis. PMC's Servo Tuning utility uses this function to capture and display servo performance. This function supports capturing:

- Actual Position versus time
- Optimal Position versus time
- Following error versus time

When initiated by the ***MCCaptureData*** function the DCX-PCI100 will perform one Motion Data Capture operation every millisecond. If more than one DCX motor module is installed, the period between data captures for the target axis will be:

1 msec. X # of installed modules

For example if 6 motor modules are installed, and the MCCaptureData function is called for axis #1, motor data will be captured for axis #1 every 6 msec's.

1 msec. X 6 modules = 6 msec's

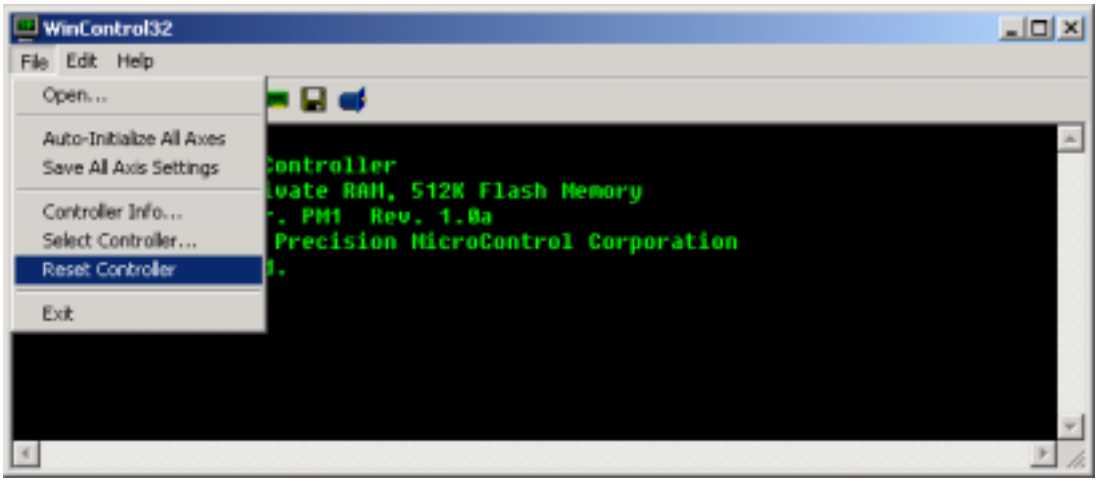
The time base for capturing data is the 1 millisecond. The function ***MCGetCapturedData( )*** is used to retrieve the captured data. The following example captures 1000 data points, then reads the captured data into an array for further processing.

```
double Data[1000];

MCBlockBegin( hCtlr, MC_BLOCK_COMPOUND, 0 );
MCCaptureData( hCtlr, 1, 1000, 0.001, 0.0 );
MCMoveRelative( hCtlr, 1, 1000.0 );
MCWaitForStop( hCtlr, 1, 0.0 );
MCBlockEnd( hCtrlr, NULL );

//  Retrieve captured actual position data into local array
//
if (MCGetCaptureData( hCtlr, 1, MC_DATA_ACTUAL, 0, 1000, &Data ) {
   . . .        // process data
```

Resetting the DCX

The DCX supports software controlled reset. To reset the DCX-PCI100 motherboard and all installed axes issue the MCAPI function ***MCReset( )***. For additional information please refer to the **MCAPI function descriptions** later in this manual.

Most PMC application programs (Motor Mover, Servo Tuning, Wincontrol) allow the user to reset the controller by selecting ***Reset Controller*** from the WinControl File menu.



Resetting the DCX-PCI100 from a user application program (with ***MCReset( )***) or from one of a PMC's software programs (by selecting ***Reset Controller*** from: Motor Mover, WinControl, Servo Tuning, etc...) will cause the controller to revert to default settings (PID, velocity, accel/decel, limits, etc...). For additional information on restoring user defined settings please refer to the Motion Control Dialog function ***MCDLG_RestoreAxis***.

In the event of a 'hang up' of the application program and/or controller, the application program may fail to resume operation after issuing the ***MCReset( )*** function. The user will have to terminate and then re-open the application program.

Until the DCX has fully re-initialized the Reset Relay (connector J5 pins 2 and 4) will be energized.

Single Stepping MCCL Programs

While the DCX is executing any Motion Control Command Language (MCCL) macro program, the user can enable single step mode by entering <ctrl> <B>. Each time this keyboard sequence is entered, the next MCCL command in the program sequence will be executed. The following macro program will be used for this example of single stepping:

```
MD10,WA1,1MR1000,1WS.1,1TP,1MR-1000,1WS.1,1TP,RP
```

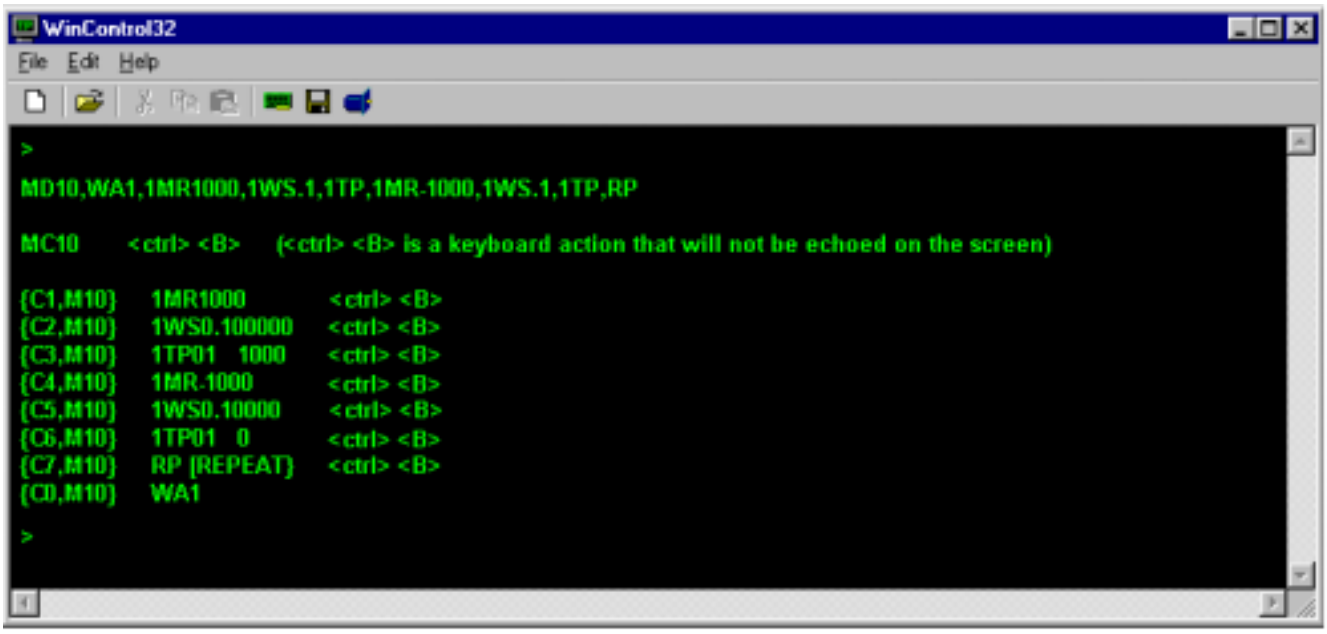
This sample program will: wait for 1 second, move 1000 encoder counts, report the position 100 msec's after the calculated trajectory is complete, move -1000 encoder counts, report the position 100 msec's after the calculated trajectory is complete, repeat the command sequence.

This command sequence can be entered directly into the memory of the DCX by typing the command sequence in the terminal interface program WinCtl32.exe or by downloading a text file via WinControl's file menu.

To begin single step execution of the above example macro enter MC10 (call macro #10) then <ctrl> <B> the following will be displayed:

{C1,MC10} 1MR1000 <

The display format of single step mode is: {Command #,Macro #} Next command to be executed



To end single stepping and return to immediate MCCL command execution press <Enter>. To abort the MCCL program enter <Escape>. Single step mode is not supported for a MCCL sequence that is executing as a background task.

Defining User Units

When power is applied or the DCX is reset, it defaults to encoder counts as its units for motion command parameters. If the user issues a move command to a servo with a target of 1000, the DCX will move the servo 1000 encoder counts. In many applications there is a more convenient unit of measure than the encoder counts of the servo. If there is a fixed ratio between the encoder counts and the desired 'user units', the DCX can be programmed with this ratio and it will perform conversions implicitly during command execution.

Defining user units is accomplished with the function ***MCSetScale( )*** which uses the **MCSCALE** data structure. This function provides a way of setting all scaling parameters with a single function call using an initialized **MCSCALE** structure. To change scaling, call ***MCGetScale( )***, update the **MCSCALE** structure, and write the changes back using ***MCSetScale( )***.

MCScale Data Structure

```
typedef struct {


double    Constant;       // Define output constant
   double    Offset;             // Define the work area zero
   double    Rate;               // Define move (vel., accel, decel) time
units
   double    Scale;          // Define encoder scaling
   double    Zero;               // Define part zero
   double    Time;               // Define time scale

} MCMOTION;
```

Setting Move (Encoder) Units

The value of the **Scale** member is the number of encoder counts per user unit. For example, if the servo encoder on axis 1 has 1000 quadrature counts per rotation, and the mechanics move 1 inch per rotation of the servo, then to setup the controller for user units of inches:

```
MCSCALE Scaling;

MCGetScale( hCtlr, 3, &Scaling );
Scaling.Scale = 1000.0;                 // 1000 encoder counts/inch
MCSetScale( hCtlr, 3, &Scaling );
```

Prior to issuing the **Scale** member, the parameters to all motion commands for a particular axis are rounded to the nearest integer. After setting a new encoder scale and calling **MCEnableAxis( )** to initialize the axis, motion targets are multiplied by the ratio prior to rounding to determine the correct encoder position. Calling the ***MCGetPosition( )*** will load the scaled encoder position.

Note – setting a user scale other than 1:1 will also scale trajectory settings (Velocity, acceleration/deceleration) but not PID settings.

Trajectory Time Base

The value of the **Rate** member sets the time unit for velocity, acceleration/deceleration values, to a time unit selected by the user. If velocities are to be in units of inches per minute, the user time unit is a minute. The value of the **Rate** member is the number of seconds per 'user time unit'. If the velocity and accel/decel are to be specified in units of inches per minute and inches per minute per minute for axis 1, then the **Rate** value should be set to 60 seconds/1 minute = 60 (1UR60). The function **MCEnableAxis( ) must** be issued before the user rate will take effect.

```
MCSCALE Scaling;

MCGetScale( hCtlr, 3, &Scaling );
Scaling.Rate = 60.0;                    // set rate to inches per minute
MCSetScale( hCtlr, 3, &Scaling );
```

Typical Rate values

Time Unit	***User Rate Conversion***
second	1 (default)
minute	60
hour	3600

Defining the Time Base for Wait commands

For the ***MCWait( )***, ***WaitForStop( )*** and ***WaitForTarget( )*** functions, the default units are seconds. By setting the member **Time**, these three commands can be issued with parameters in units of the user's preference. The parameter to member is the number of 1 second periods in the user's unit of time. If the user prefers time parameters in units of minutes, **Time** = 60 should be issued.

```
MCSCALE Scaling;

MCGetScale( hCtlr, &Scaling );
Scaling.Time = 60.0;                       // set Wait time unit to minutes
MCSetScale( hCtlr, &Scaling );
```

Defining a System/Machine zero

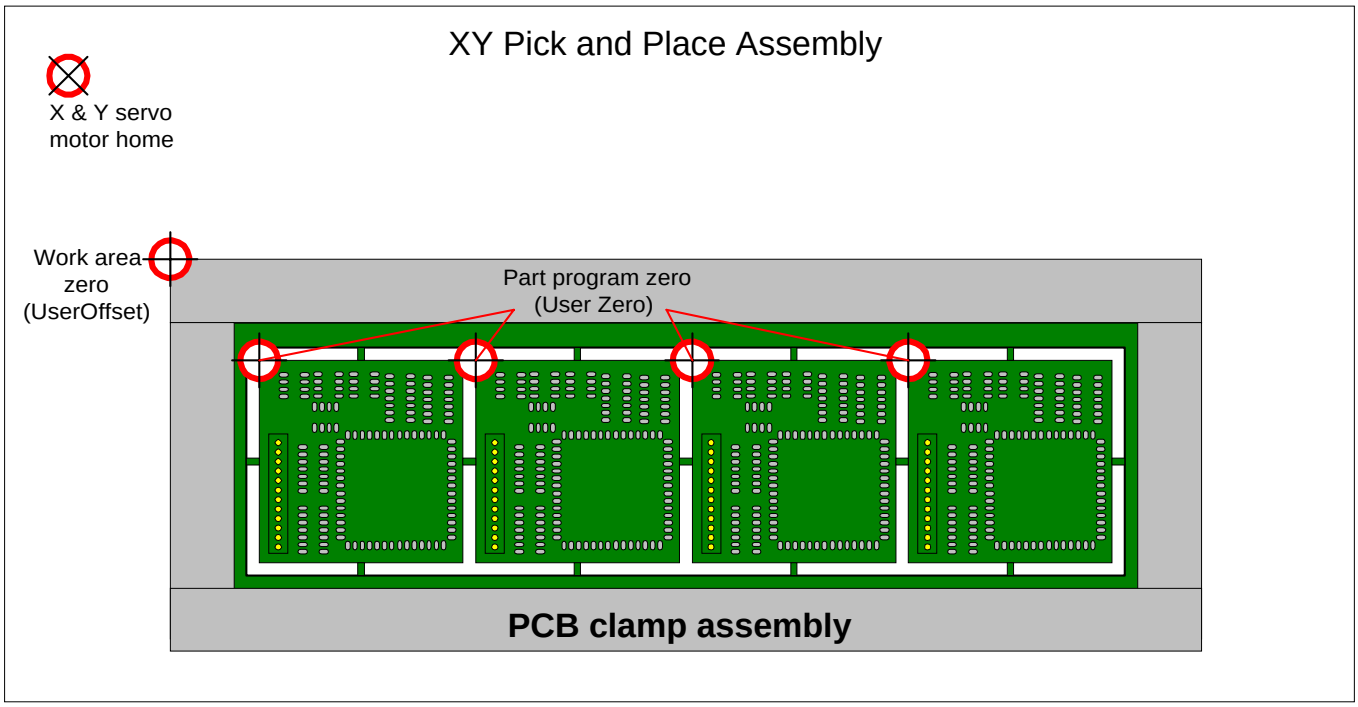
The member **Offset** allows the user to define a ‘work area’ zero position of the axis. The **Offset** value should be the distance from the servo motor home position, to the machine zero position. This offset distance must use the same units as currently defined by set User Scaling command. **Offset** does not change the index or home position of the servo motor, it only establishes an arbitrary zero position for the axis.

```
MCSCALE Scaling;

MCGetScale( hCtlr, 3, &Scaling );
Scaling.Offset = 12.25;                 // define offset to 12.25 inches
MCSetScale( hCtlr, 3, &Scaling );
```

Defining a Part Zero

The member **Zero** would typically be used in conjunction with **Offset** to define a ‘part zero’ position. A PCB (Printed Circuit Board) pick and place operation is a good example of how this function would be used. After a new PCB is loaded and clamped into place the X and Y axes would be homed. The **Offset** member is used to define the ‘work area’ zero of the PCB. The Zero member is used to define the ‘part program’ or ‘local’ zero position. This way a single ‘part placement program’ can be developed for the PCB type, and a ‘step and repeat’ operation can be used to assemble multiple part assemblies.

```
MCSCALE Scaling;

MCGetScale( hCtlr, 3, &Scaling );
Scaling.Offset = 12.25;                 // define offset to 12.25 inches
Scaling.Zero = 1.25;                    // define 'part zero' to 1.25 inches
MCSetScale( hCtlr, 3, &Scaling );
```



DCX Watchdog

The DCX incorporates a watchdog circuit to protect against improper CPU operation.

After a reset or power cycle, once the firmware (operational code) has been loaded by the operating system (approximately 3 seconds), the watchdog circuit is enabled.

If the DCX processor fails to properly execute firmware code for a period of 10 msec's, the watchdog circuit will 'time out' and the on-board reset will be latched by the ‘watchdog reset relay’. This in turn will hold the DCX modules in a constant state of reset. All motor command/drive outputs will be disabled. When the watchdog circuit has tripped, the green **Run LED** will be disabled. To clear the watchdog error either:

Cycle power to the computer ***(recommended)***
Reset the computer

Note: If the watchdog trips while a MCAPI based application program is running, manually resetting the DCX will **probably not** allow the application program to continue operation.

Chapter Contents

Chapter 9

General Purpose I/O

DCX Motherboard Digital I/O

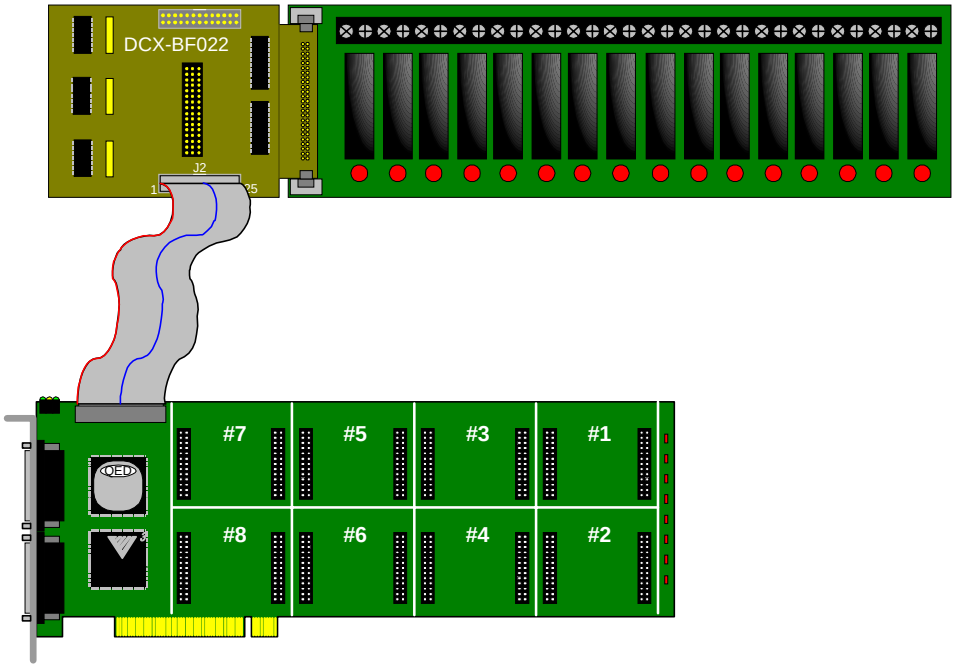
The DCX-PCI100 Motion Controller motherboard has 16 general purpose digital I/O channels. Channels 1 – 8 are TTL inputs and channels 9 – 16 are TTL outputs. These signals can be accessed on connector J3 of the motherboard. The **DCX-PCI100** section of the **Connectors, Jumpers, and Schematics** chapter includes a pin-out for this connector. Each digital channel is configured via software (high true or low true).

Interfacing to the 'Outside World'
The TTL digital I/O channels can be connected directly to external circuits if output loading **(1ma maximum sink/source)** and input voltages **(0.0V to +5.0V)** are within acceptable limits.

The DCX Digital I/O channels are not suitable for driving optical isolators, relays solenoids, etc...

Alternatively, a DCX-BFO22 interface board can be used to connect the module's I/O to a relay rack in order to provide optically isolated inputs and outputs.

The DCX-BFO22 interface board provides a convenient means of connecting the DCX-PCI100 TTL digital I/O channels to a 16 position relay rack available from two manufacturers, Opto22 (P/N PB16H) and Grayhill (P/N 70RCK16-HL). These relay racks accept up to 16 optically isolated input or output modules for interfacing with external electrical systems. Using one of these relay racks and a DCX-BFO22, an optically isolated I/O module can be connected to each of the DCX's digital I/O channels.



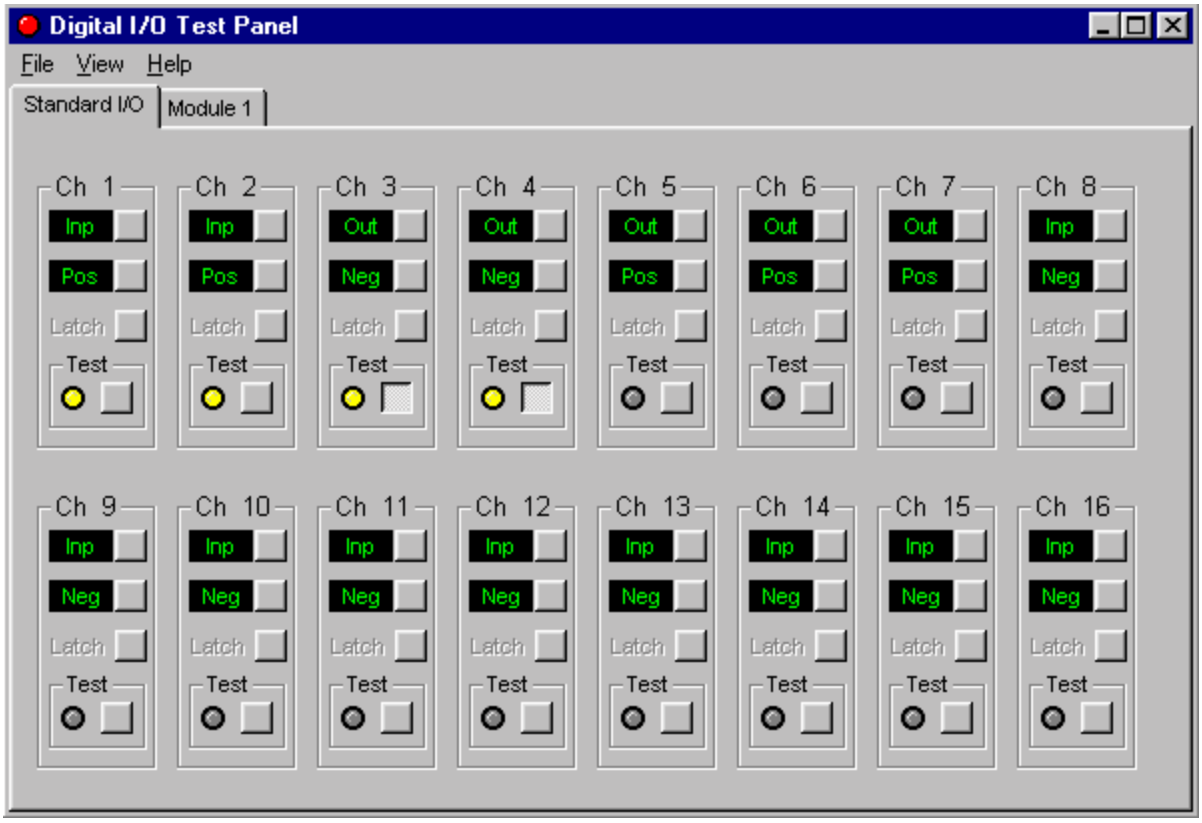
As shown above, the DCX-BFO22 plugs directly into the relay rack's 50 pin header connector and then connects to the DCX-PCI100 via a 26 conductor ribbon cable. Note that the relays are numbered sequentially starting from 0, while the DCX digital I/O channels are numbered sequentially starting with 1.

Although the relay rack has screw terminals for connecting a logic supply, it is not necessary to make this connection. By installing a shorting block on jumper JP17 of the BFO22, the 5 volt supply of the DCX will be supplied to the relay rack.

For detailed information on configuring the DCX-BF022, please refer to the schematic and jumper table in the **Connectors, Jumpers, and Schematic** chapter later in this manual.

Configuring the DCX Digital I/O

The configuration of both the DCX-PCI100 and the DCX-MC400 digital I/O channels is accomplished using either PMC's Motion Integrator software or the MCAPI function ***MCConfigureDigitalIO( )***. The screen shot that follows shows the Motion Integrator Digital I/O test panel. This tool is used to both configure each I/O channel and then verify its operation. A comprehensive on-line help document is provided.



Each DCX-PCI100 digital I/O channel is individually programmable as:

High true/Positive logic (MC_DIO_HIGH) or Low true/Negative logic (MC_DIO_LOW)

Each DCX-MC400 digital I/O channel is individually programmable as:

Input (MC_DIO_INPUT) or Output (MC_DIO_OUTPUT)
High true/Positive logic (MC_DIO_HIGH) or Low true/Negative logic (MC_DIO_LOW)

The 16 channels of the DCX-PCI100 motherboard are defined as channels 1 – 16. If one or more DCX-MC400 Digital I/O modules are installed, the additional I/O channels are assigned to succeeding channel/numbers in blocks of 16 (e.g. 17-32, 33-48, etc.). All I/O channels accept the same configuration, monitoring and control.

Note – If a BFO22 interface and relay rack are connected to the DCX Digital I/O, a MC_DIO_LOW command set to ALL_AXES should be issued to the DCX. This will cause "normally open" relays to turn on when the Channel oN command is issued, and off when the Channel oFf command is issued.

This example configures all the digital I/O channels (PCI100 channels 9 – 16 and all MC400 channels) on a controller for outputs, then turns each channel on (in order) for a half second.

```
MCPARAM Param;

MCGetMotionConfig( hCtlr, &Param );

for (i = 9; i <= Param.DigitalIO; i++) {
   MCConfigureDigitalIO( hCtlr, i, MC_DIO_OUPUT | MC_DIO_HIGH );

for (i = 1; i <= Param.DigitalIO; i++) {
   MCEnableDigitalIO( hCtlr, i, TRUE );
   MCWait( hCtlr, 0.5 );
   MCEnableDigitalIO( hCtlr, i, FALSE );
}
```

Using the DCX Digital I/O

After configuring the Digital I/O channels, three MCAPI functions are available for activating and monitoring the digital I/O:

MCEnableDigitalIO( )	set digital output channel state
MCGetDigitalIO( )	get digital input channel state
MCWaitForDigitalIO( )	wait for digital input channel to reach specific state

Enable Digital IO

Turns the specified digital I/O on or off, depending upon the value of *bState*.

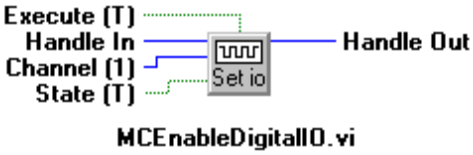
TRUE Turns the channel on.
FALSE Turns the channel off.

The I/O channel selected must have previously been configured for output using the ***MCConfigureDigitalIO( )*** command. Note that depending upon how a channel has been configured "on" (and conversely "off") may represent either a high or a low voltage level.

compatibility:	MC400
see also:	Configure Digital IO

C++ Function:	void MCEnableDigitalIO(HCTRLR hCtlr, WORD wChannel, short int bState);
Delphi Function:	procedure MCEnableDigitalIO(hCtlr: HCTRLR; wChannel: Word; bState: SmallInt);
VB Function:	Sub MCEnableDigitalIO (ByVal hCtrlr As Integer, ByVal channel As Integer, ByVal state As Integer)
MCCL command:	CF, CN

LabVIEW VI:



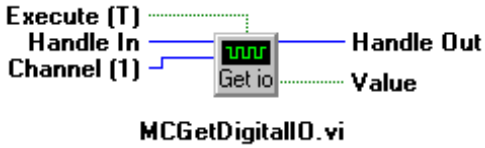
Get Digital IO

Returns the current state of the specified digital I/O channel. This function will read the current state of both input and output digital I/O channels. Note that this function simply reports if the channel is "on" or "off"; depending upon how a channel has been configured "on" (and conversely "off") may represent either a high or a low voltage level.

compatibility: MC400
see also:

C++ Function: short int MCGetDigitalIO(HCTRLR hCtlr, WORD wChannel);
Delphi Function: function MCGetDigitalIO(hCtlr: HCTRLR; wChannel: Word): SmallInt;
VB Function: Function MCGetDigitalIO (ByVal hCtrlr As Integer, ByVal channel As Integer) As Integer
MCCL command : TC

LabVIEW VI:



Wait for Digital IO

Waits for the specified digital I/O channel to go on or off, depending upon the value of bState.

compatibility: MC400
see also: Wait for digital channel on

C++ Function: void MCWaitForDigitalIO(HCTRLR hCtlr, WORD wChannel, short int bState);
Delphi Function: procedure MCWaitForDigitalIO(hCtlr: HCTRLR; wChannel: Word; bState: SmallInt);
VB Function: Sub MCWaitForDigitalIO (ByVal hCtrlr As Integer, ByVal channel As Integer, ByVal state As Integer)
MCCL command: WF, WN

LabVIEW VI:

This example configures all the digital I/O channels on a controller for output, then turns each channel on (in order) for a half second.

DCX Module Analog I/O

The DCX-MC500 Analog I/O Module provides analog I/O capability for a DCX Motion Controller. One or more of these modules can be installed in any available module position on a DCX motherboard. Analog input channels can be used to monitor signal levels from external sensors. Output channels can be used to control external devices.

Three models of the DCX-MC500 are available:

Part Number	***Description***
DCX-MC500	4 Inputs and 4 Outputs
DCX-MC510	4 Inputs
DCX-MC520	4 Outputs

On each DCX-MC500/510 Analog I/O Module all analog input channels are numbered sequentially in groups of four. Likewise, all analog output channels are numbered sequentially in groups of four. When installed on the DCX-PCI100, the MC500/510 in the **lowest module location** will have its 4 analog input channels defined as 1 – 4. The four analog inputs of a MC500/510 installed in the next lowest module location will be defined as channels 5 – 8.

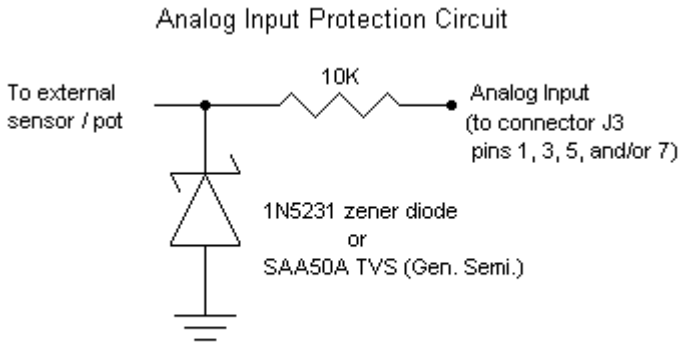
Because the DCX controller board is implemented in digital electronics, all analog input signals must be converted into a representative numerical value. This function is done by an Analog to Digital Converter (ADC) on the DCX-MC500/510. Similarly, analog output signals originate on the DCX board as numerical values. These numbers must be written to a Digital to Analog Converter (DAC) on the DCX-MC500/520, which converts them to a corresponding analog output signal level.

The DCX-MC500 is designed to accurately measure voltage levels on the input channels. These inputs are very high impedance with leakage currents less than 10 nano amps. The output channels are designed to provide signals with accurate voltage levels. The current requirement from these outputs should not exceed ***10 milliamps***.

Each of the analog input and analog output channels has 12 bits of resolution. This means that the digital value read from the ADC, or the digital value written to DAC, must be in the range 0 to 4095. For both inputs and outputs, a digital value of 0 translates to the lowest analog voltage. A digital value of 4095 translates to the highest analog voltage.

Input signals on pins 1, 3, 5 and 7 of the module J3 connector are wired directly to the ADC. No amplification or clamping to the input voltage range is provided on the module.

A voltage level greater than 5.6 volts will damage the analog input channels of a DCX-MC5X0 module. The schematic below is recommended to protect an analog input from damage due to an over voltage condition. This circuit will limit the maximum voltage applied to the A/D converter to 5.6 VDC.



In some applications, the signals from a sensor may not be absolute voltage levels, but proportional to some reference voltage. In these cases, it may be desirable to supply the reference signal to the ADC on the module through pin 18 of the J3 connector (and setting jumper JP1 accordingly). This will result in a "ratiometric" conversion of the input signal relative to the reference voltage.

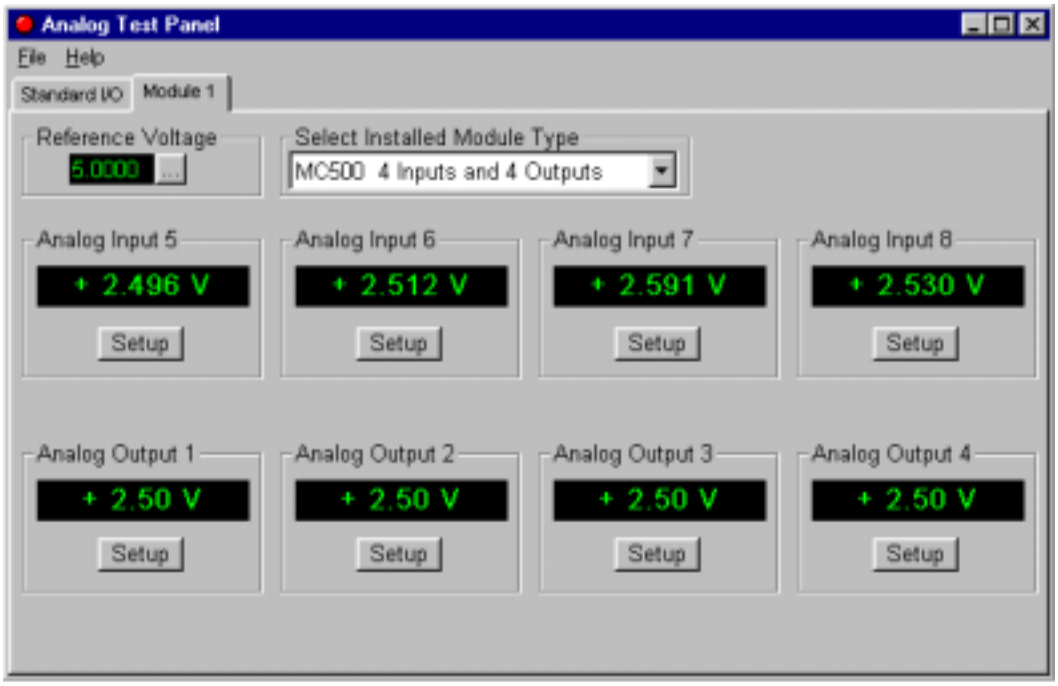
The outputs from the DAC on the DCX-MC500 module are voltage levels in the range 0 to +5 volts. These outputs have no gain or offset adjustment. These signals are available on pins 10, 12, 14 and 16 of the module J3 connector.

The outputs from the DAC are also connected to operational amplifiers on the module, which offset and amplify them to provide a +/-10 volt range. Each of these outputs has a 20 turn trim pot for offset adjustment, and a single turn pot for gain adjustment. The offset pot provides a minimum 0.5 volt adjustment, and the gain pot provides a nominal 2% range adjustment. These output signals are available on pins 2, 4, 6 and 8 of the module J3 connector.

After reset the outputs of the DCX-MC500 will be initialized to their mid-scale point. For the 0 to +5 volt outputs, this will be ***2.5 volts***. For the -10 to +10 volt outputs, this will be ***0.0*** volts.

Using the Analog I/O

The configuration and operation of the DCX-MC5X0 analog I/O channels is accomplished using either PMC's Motion Integrator program or the MCAPI functions ***MCSetAnalog( )*** , ***MCGetAnalog( )***. The screen capture that follows shows the Motion Integrator Analog I/O test panel. This tool is used to both configure each I/O channel and then verify its operation. A comprehensive on-line help document is provided.



Two MCAPI functions are available for setting and monitoring the MC500 analog I/O:

MCSetAnalog( )	set digital output channel state
MCGetAnalogIO( )	get digital input channel state

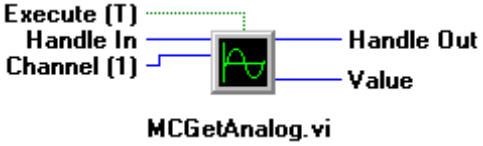
Get Analog

Reads the digitized input state of the specified input *wChannel*. The four 8-bit analog input channels accessed on connectors J3 are numbered 1,2,3 and 4. For each of these channels, this function will read a number between 0 and 255. These numbers are the ratio of the analog input voltage to the reference input voltage multiplied by 256. The reference voltage for the first four channels must be supplied to the DCX on the J3 connector pin 23, and can be any voltage between 0 and +5 volts DC. The analog input channels on any installed MC500 modules will be numbered sequentially starting with channel 5. See the description of **Analog Inputs** in the **DCX General Purpose I/O** chapter.

compatibility: MC500, MC510
see also: Set Analog

C++ Function: WORD MCGetAnalog(HCTRLR hCtlr, WORD wChannel);
Delphi Function: function MCGetAnalog(hCtlr: HCTRLR; wChannel: Word): Word;
VB Function: Function MCGetAnalog (ByVal hCtrlr As Integer, ByVal channel As Integer) As Integer
MCCL command: TA

LabVIEW VI:



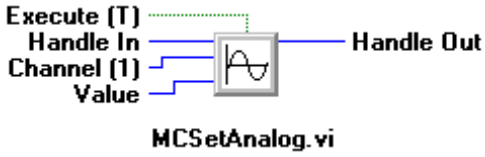
Set Analog

Sets the output level of an analog channel. Analog output ports on MC500 and MC520 Analog Modules accept values in the range of 0 to 4095 counts (12 bits). This range of values corresponds to an output voltage of 0 to 5V or -10 to +10V, depending upon how the output is configured (See the description of **Analog Inputs** in the **DCX General Purpose I/O** chapter).

compatibility: MC500, MC520
see also: Get Analog

C++ Function: void MCSetAnalog(HCTRLR hCtlr, WORD wChannel, WORD wValue);
Delphi Function: procedure MCSetAnalog(hCtlr: HCTRLR; wChannel, value: Word);
VB Function: Sub MCSetAnalog (ByVal hCtrlr As Integer, ByVal channel As Integer, ByVal Value As Integer)
MCCL command: OA

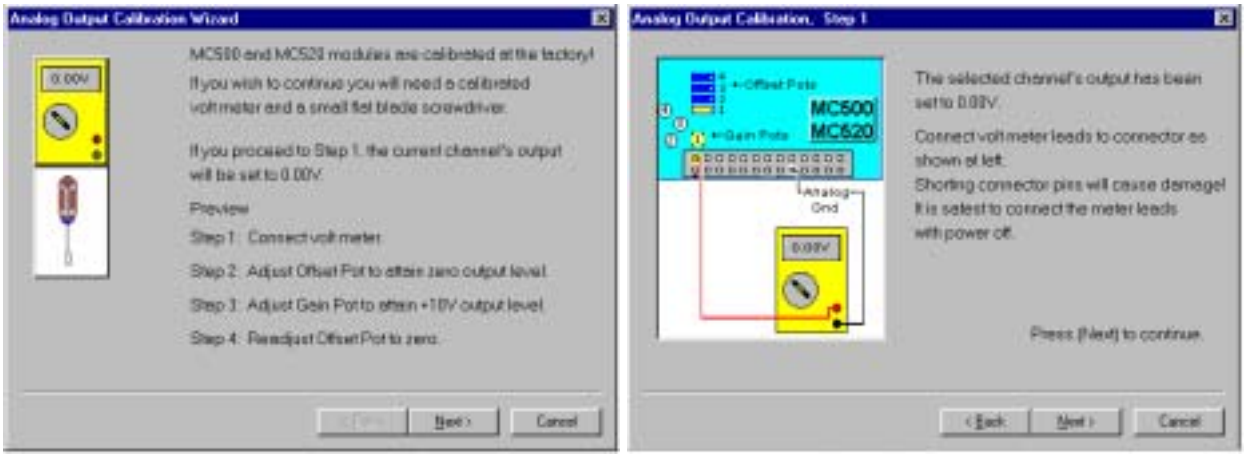
LabVIEW VI:



Calibrating the MC500/MC520 +/- 10V Analog Outputs:

The analog inputs of the DCX-MC500 require no calibration, and the only option is use of the internal +5, or an external, reference voltage. The analog outputs with the 0 to +5 volt range also have no adjustments. The reference for the DAC is fixed to the internal reference voltage.

The four 0.0 to +5.0 analog outputs require no calibration. The four +10 to –10 volt analog outputs are calibrated at the factory. There are four single turn trim pots that are used to adjust the gain of each of the four analog outputs. There are also four 20 turn trim pots for adjusting the offsets of each of the analog outputs. It is **strongly recommended** that the +10 to –10 volt outputs be calibrated using the **Motion Integrator Calibration Wizard**.



The analog outputs can also be calibrated using MCCL command sequences. For a description of **MCCL** commands and the **WinControl command interface utility** please refer to the MCCL section of the appendix at the end of this user manual. Refer to the module layout diagram in the **Connectors, Jumpers, and Schematics** chapter of this user manual. Using the following command sequence, and reading the analog output voltage level with a voltmeter, an analog output can be calibrated to provide the specified -10 to +10 volt range:

```
AL0,OAn,WA2,AL2048,OAn,WA2,AL4095,OAn,WA2,RP
```

where: n = channel number = 1, 2, 3, 4, ...

This command sequence will cycle the specified analog output from the minus limit, to the mid-point, to the positive limit. There is a 2 second delay at each voltage level, during which the voltmeter can settle and display the current reading.

The first step in calibrating an analog output is to adjust the gain using the single turn pot to achieve a 20.00 volt "swing". This is the difference between the most positive level reading, and the most negative level reading. It is not necessary for the two readings to be centered about 0 volts for this step.

The second step is to adjust the offset using the 20 turn pot. This adjustment will place the mid-point of analog output at the 0 volt level. When the output changes to the mid- point level turn the pot to achieve a 0.000 volt reading.

After the second step of the calibration procedure, the output swing should still be 20.00 volts. If not, repeat steps 1 and 2 again.

Chapter Contents

Chapter 10

Motion Control API Introduction

The Motion Control Application Programming Interface (MCAPI) implements a powerful set of high level functions and data structures for programming motion control applications. Although this manual has been written for the latest version of the MCAPI software, there are still remnants of deprecated functions. The older functions will still work with this version, however, we recommend that the newer functions be migrated to when feasible.

The API is backwards compatible, and applications may use the most current version of the MCAPI for products of varying generations. Care must be taken to note the exceptions of newer features that older products might not be capable of utilizing, as well as older functions may not be relevant to new controllers. Please observe the compatibility section in each function.

Function Listing Introduction

An example of a function listing is shown below. What follows the example is a brief description of what should be found in each of the respective headings.

MCEnableAxis

MCEnableAxis( ) turns the specified axis on or off.

```
void MCEnableAxis(
   HCTRLR hCtlr,                 // controller handle
   WORD axis,                    // axis number
   short int state               // Boolean flag for on/off setting of axis
);
```

Parameters

hCtlr Controller handle, returned by a successful call to **MCOpen( )**.

axis Axis number to turn on or off.
state Flag to indicate if this axis should be turned on or turned off:

Value	Description
TRUE	Turn on *axis*.
FALSE	Turn off *axis*.

Returns
This function does not return a value.

Comments
This function does much more than just enable or disable *axis*. However, as the name implies, the selected axis(axes) will be turned on or off depending upon the value of *state*. Note that an axis must be enabled before any motion will take place. Issuing this command with *axis* set to MC_ALL_AXES will enable or disable all axes installed on *hCtlr*.

state will accept any non-zero value as TRUE, and will work correctly with most programming languages, including those that define TRUE as a non-zero value other than one (one is the Windows default value for TRUE).

If *axis* is off and then turned on, the following events will occur.

- The target and optimal positions are set to the present encoder position.
- The offset from **MCFindEdge( )**, **MCFindIndex( )** or **MCIndexArm( )** is applied.
- The data passed by **MCSetScale( )** are applied.
- MC_STAT_AMP_ENABLE will be set.
- MC_STAT_AMP_FAULT, if present, will be cleared.
- MC_STAT_ERROR, if present, will be cleared.
- MC_STAT_FOLLOWING, if present, will be cleared.
- MC_STAT_MLIM_TRIP, if present, will be cleared.
- MC_STAT_MSOFT_TRIP, if present, will be cleared.
- MC_STAT_PLIM_TRIP, if present, will be cleared.
- MC_STAT_PSOFT_TRIP, if present, will be cleared.

If *axis* is on and then turned on again, the following events will occur.

- The offset from **MCFindEdge( )**, **MCFindIndex( )** or **MCIndexArm( )** is applied.
- The data passed by **MCSetScale( )** are applied.

Calling this function to enable or disable an axis while it is in motion is not recommended. However, should it be done, *axis* will cease the current motion profile, and MC_STAT_AT_TARGET will be set.

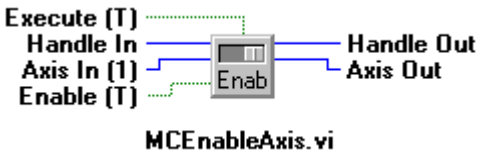
Compatibility
There are no compatibility issues with this function.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure MCEnableAxis(hCtlr: HCTRLR; axis: Word; state: SmallInt); stdcall;
VB: Sub MCEnableAxis (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal state As Integer)
LabVIEW:



MCCL Reference

MF, MN

See Also

MCAbort( ), **MCStop( )**

Each function definition begins with a brief introductory description that explains what the function is used for.

Following the description, a grey box contains the C/C++ function prototype. Here each of the parameters is listed with its type and a short description for a quick overview.

Parameters then further explains in more detail what each of the parameters means. Here a table, if applicable, will be included listing the allowable values for the preceding parameter. When values are listed, they will be given as self documenting constants. A complete listing of the self documenting constants can be found in Appendix B.

Returns describes what the function will return and explains what those values mean. The self documenting constants will be referenced when possible.

Comments describes the function in even more detail. Explanation will range from why the function is used, to how it is used, where it could cause problems and potential alternatives.

Occasionally, the following two boxes can be found in the comments section and contain relevant information that needs to be emphasized. The first box aids in the understanding of the function. The second box warns of scenarios that will more than likely cause problems.

Information to assist the programmer.

Warning to help the programmer avoid potential problems.

Compatibility gives information as to which motion control cards or modules will not work with the function. Generally, only exceptions will be listed, as to provide a more concise listing.

Requirements lists which header files, library, and the MCAPI version that must be used. Obviously, only the header file which pertains to the development environment must be used. The version of the MCAPI that is referenced is the earliest version that supports the function, so any version higher that is used will not cause a problem.

Prototypes lists the function prototypes for Delphi/Pascal, Visual Basic, and LabVIEW. As shown, each of the parameters are listed with their type. Not all functions will be available in all environments and will be noted as “Not Supported” when exceptions exist.

MCCL Reference lists the MCCL level commands that comprise the high level function. More information can be found in the **Motion Control Command Language (MCCL) Reference Manual** specific to your controller on how each of these commands works. Not all functions will be comprised of speaking to the board with MCCL commands, in which cases there will be no equivalent commands.

See Also lists related functions. Some of these functions may be alternatives to be used, while others may be the corresponding get function to a set function. Yet there will be other functions that must be used as in tandem with another function.

Motion Control API Function Quick Reference Tables

The following tables show how functions have been classified categorically. Although several functions could quite logically be listed in multiple categories, each function will appear in only one chapter, which is noted by the table's heading. The organization follows closely to prior manuals and the online help. The grouping of functions in this manner gives a new user of the MCAPI software a chance to find similar functions in one place. For a handy quick reference printout, please refer to the **MCAPI Quick Reference Card**, which can be found on our website (www.pmccorp.com) under support and then Motion Control API. The quick reference card lists all of the following functions, as well as the data structures and the constants, in a convenient, alphabetical listing.

Parameter Setup Functions

Function	Description
MCConfigureCompare( )	configure high-speed position compare
MCSetAcceleration( )	set Acceleration for an axis
MCSetAuxEncPos( )	set the position of the auxiliary encoder
MCSetCommutation( )	configure commutation
MCSetContourConfig( )	set contour configuration settings
MCSetDeceleration( )	set deceleration for an axis
MCSetDigitalFilter( )	configure digital filter
MCSetFilterConfigEx( )	set the PID filter parameters
MCSetGain( )	set the proportional gain for a servo axis
MCSetJogConfig( )	set jogging configuration for axis
MCSetLimits( )	configure hard and soft limits for an axis
MCSetModuleInputMode( )	configure stepper module input mode
MCSetModuleOutputMode( )	define the output type
MCSetMotionConfigEx( )	set motion parameters (velocity, accel, step rate, dead band, etc...)
MCSetOperatingMode( )	set the mode of motion (position, velocity, contour, torque)
MCSetPosition( )	set the current position of an axis
MCSetProfile( )	select a motion profile (trapezoidal, s-curve, parabolic)
MCSetRegister( )	set general purpose user register
MCSetScale( )	set the scaling factors for an axis
MCSetServoOutputPhase( )	select normal or reverse phasing for a servo axis
MCSetTorque( )	set output voltage limit for servo
MCSetVectorVelocity( )	set the vector velocity of a contoured move
MCSetVelocity( )	set the maximum velocity for a one axis move

I/O Functions

Function	Description
MCConfigureDigitalIO( )	configure digital I/O channels (input, output, high true, low true)
MCEnableDigitalIO( )	set the state of a digital output channel
MCGetAnalog( )	read analog input channel value
MCGetDigitalIO( )	get the state of a digital input channel
MCGetDigitalIOConfig( )	get digital I/O channel configuration
MCSetAnalog( )	set the value of an analog output
MCWaitForDigitalIO( )	wait for digital I/O channel to reach a specific state

Macro's and Multi-Tasking Functions

Function	Description
MCCancelTask( )	cancel a background task
MCMacroCall( )	call a MCCL macro
MCRepeat( )	inserts a repeat command into a macro or task sequence

Motion Functions

Function	Description
MCAbort( )	abort the current motion for an axis
MCArcCenter( )	sets the center point of an arc
MCArcEndAngle( )	defines the ending angle of an arc
MCArcRadius( )	defines the radius of an arc
MCCaptureData( )	initiate real time capture of position and servo loop data
MCContourDistance( )	set the path distance for user defined contour motion
MCDirection( )	set travel direction for velocity mode move
MCEdgeArm( )	arm edge input for position capture
MCEnableAxis( )	turn axis on or off
MCEnableBacklash( )	enable backlash compensation
MCEnableCapture( )	enable position capture
MCEnableCompare( )	enable position compare
MCEnableDigitalFilter( )	enable digital filter
MCEnableGearing( )	enable/disable gearing
MCEnableJog ( )	enable/disable jogging for axis
MCEnableSync( )	enables cubic spline motion, synchronizes contour motion
MCFindAuxEncIdx( )	initialize the auxiliary encoder at the location of the index
MCFindEdge( )	initialize a stepper motor at the location of the home input
MCFindIndex( )	initialize a servo motor at the location of the encoder index input
MCGoEx( )	start a velocity mode motion, begin cubic spline motion sequence
MCGoHome( )	move axis to absolute position 0
MCIndexArm( )	arms encoder index capture
MCLearnPoint( )	store position in point memory
MCMoveAbsolute( )	move axis to absolute position
MCMoveRelative( )	move axis to relative position
MCMoveToPoint( )	move to position stored in point memory
MCReset( )	perform a software reset of the controller
MCStop( )	stop motion
MCWait( )	wait for a variable time period
MCWaitForEdge( )	wait for the home input
MCWaitForIndex( )	wait for the index input to go true.
MCWaitForPosition( )	wait for axis to reach absolute position
MCWaitForRelative( )	wait for axis to reach relative position
MCWaitForStop( )	wait for the calculated trajectory to be complete
MCWaitForTarget( )	wait for axis to reach target position

MCAPI Driver Functions

Function	Description
MCBlockBegin( )	begin a compound commands (contour motion, macro's, multi-tasking)
MCBlockEnd( )	end a compound commands (contour motion, macro's, multi-tasking)
MCClose( )	close a controller (free handle)
MCGetConfigurationEx( )	obtain PMC controller hardware configuration
MCGetVersion( )	get the version of the DLL and device driver
MCOpen( )	open a controller (get handle)
MCReopen( )	re-opens existing controller handle for a new mode
MCSetTimeoutEx( )	set a timeout value for controller

Reporting Functions

Function	Description
MCDecodeStatus( )	axis status word decoding
MCErrorNotify( )	enables/disables error messages for application window
MCGetAccelerationEx( )	get current programmed acceleration for axis
MCGetAuxEncIdxEx( )	get last observed position of auxiliary encoder index pulse
MCGetAuxEncPosEx( )	get current position of auxiliary encoder
MCGetAxisConfiguration( )	get the axis type, location, and capabilities
MCGetBreakpointEx( )	get the most recent breakpoint position
MCGetCaptureData( )	retrieve captured axis data (current position, optimal position, error)
MCGetContourConfig( )	get contour configuration settings
MCGetContouringCount( )	get current contour count
MCGetCount( )	get count parameter of various modes
MCGetDecelerationEx( )	get current programmed deceleration for axis
MCGetDigitalFilter( )	get digital filter settings
MCGetError( )	returns the most recent controller error
MCGetFilterConfigEx( )	get the PID parameters
MCGetFollowingError( )	get the current programmed following error
MCGetGain( )	get the current proportional gain setting for an axis
MCGetIndexEx( )	get the last observed position of the primary encoder index pulse
MCGetInstalledModules( )	Enumerates the type of DCX modules
MCGetJogConfig( )	get jogging configuration for axis
MCGetLimits( )	get current hard and soft limit settings
MCGetModuleInputMode( )	get the current input mode for a stepper module
MCGetMotionConfigEx( )	get motion configuration
MCGetOperatingMode( )	get the current operating mode for a motor module
MCGetOptimalEx( )	get the current optimal position of an axis
MCGetPositionEx( )	get the current position of an axis
MCGetProfile( )	get the current profile type (trapezoidal, s-curve, parabolic)
MCGetRegister( )	get the contents of a general purpose register
MCGetScale( )	get the current programmed scaling factors for an axis
MCGetServoOutputPhase( )	get the output phase (normal or reversed) of a servo
MCGetStatus( )	get the axis status word
MCGetTargetEx( )	get the current target of an axis
MCGetTorque( )	get the current torque setting of an axis
MCGetVectorVelocity( )	get the current programmed vector velocity of an axis
MCGetVelocityEx( )	get the current programmed velocity of an axis
MCIsAtTarget( )	is axis at target position?
MCIsDigitalFilter( )	is digital filter enabled?
MCIsEdgeFound( )	has edge input gone true?
MCIsIndexFound( )	has index pulse been found?
MCIsStopped( )	is axis stopped?
MCTranslateErrorEx( )	translate numeric error code to text message

OEM Low Level Functions

Function	Description
pmccmd( )	send a binary command
pmccmdex( )	send a binary command
pmcgetc( )	get ASCII character from controller
pmcgetram( )	read directly from controller memory
pmcgets( )	get ASCII string from controller
pmcputc( )	write ASCII character to controller
pmcputram( )	write directly to controller memory
pmcputs( )	write ASCII string to controller
pmcrdy( )	is the controller ready to accept a binary command
pmcrpy( )	read binary reply from controller
pmcrpyex( )	read binary reply from controller

Motion Dialog Functions

Function	Description
MCDLG_AboutBox( )	display a simple About dialog box
MCDLG_CommandFileExt( )	get the file extension for MCCL command files
MCDLG_ConfigureAxis( )	display a servo or stepper axis setup dialog
MCDLG_ControllerDescEx( )	get a descriptive string for a motion controller type
MCDLG_ControllerInfo( )	get configuration information about a motion controller
MCDLG_DownloadFile( )	download an ASCII command file to a motion controller
MCDLG_Initialize( )	must be called before any other MCDLG functions or classes
MCDLG_ListControllers( )	get the types of motion controllers installed
MCDLG_ModuleDescEx( )	get a descriptive string for a module
MCDLG_RestoreAxis( )	restore the settings of an axis to a previously saved state
MCDLG_RestoreDigitalIO( )	restores the settings of digital I/O channels to previously saved states
MCDLG_SaveAxis( )	save the settings of an axis to an initialization file for later use
MCDLG_SaveDigitalIO( )	save the settings of digital I/O channels to an initialization file
MCDLG_Scaling( )	display a scaling setup dialog and allow changes to scaling parameters.
MCDLG_SelectController( )	display a list of installed controllers and allow selection of a controller

Chapter Contents

Chapter 11

Data Structures

The following data structures allow the programmer to pass data to and from the controller in a simple and efficient manner. Structures are the only way, short of using MCCL, to set and get certain parameters to and from the motion control card. Functions listed in the "see also" section rely on these data structures. The chapters on Parameter Setup Functions and Reporting Functions contain the majority of the functions that require these structures.

MCAXISCONFIG

MCAXISCONFIG structure provides basic information about the type and configuration of a single motor axis.

```
typedef struct {
   long int cbSize;
   long int ModuleType;
   long int ModuleLocation;
   long int MotorType;
   long int CaptureModes;
   long int CapturePoints;
   long int CaptureAndCompare;
   double HighRate;
   double MediumRate;
   double LowRate;
   double HighStepMin;
   double HighStepMax;
   double MediumStepMin;
   double MediumStepMax;
   double LowStepMin;
   double LowStepMax;
} MCAXISCONFIG;
```

Members

cbSize Size of the **MCAXISCONFIG** data structure, in bytes.
ModuleType Array of OEM axis type specifiers, one per axis:

Value	Description
MC100	Identifies a DC Servo axis with analog signal output.
MC110	Identifies a DC Servo axis with motor output.
MC150	Identifies a stepper motor axis.
MC160	Identifies a stepper motor with encoder axis.
MC200	Identifies an Advanced Servo axis with analog signal output.
MC210	Identifies an Advanced Servo axis with PWM motor output.
MC260	Identifies an Advanced Stepper axis.
MC300	Identifies a DSP-Based Servo axis with analog signal output.
MC302	Identifies a DSP-Based Dual Servo axes with dual analog signal outputs.
MC320	Identifies a DSP-Based Brushless AC Servo axis with dual analog signal outputs.
MC360	Identifies a DSP-Based Stepper axis.
MC362	Identifies a DSP-Based Dual Stepper axes.
MF300	Identifies this axis as an RS-232 communications module. This module is not normally used with a controller installed in a PC adapter slot.
MF310	Identifies this axis as an IEEE-488 (GPIB) communications module. This module is not normally used with a controller installed in a PC adapter slot.
MC400	Identifies this axis as providing additional digital I/O channels (16).
MC500	Identifies this axis as providing additional analog channels.
DC2SERVO	Identifies the dedicated servo output of a DC2 controller.
DC2STEPPER	Identifies the optional stepper output of a DC2 controller.

MotorType Provides a simplified type identifier for the motor type (bit flags):

Value	Description
MC_TYPE_SERVO	Axis is a servo motor.
MC_TYPE_STEPPER	Axis is a stepper motor.

CaptureModes Supported data capture modes for this axis (bit flags). One or more of the following values may be OR'ed together:

Value	Description

Value	Description
MC_CAPTURE_ACTUAL	Axis can capture actual position data.
MC_CAPTURE_ERROR	Axis can capture error position data.
MC_CAPTURE_OPTIMAL	Axis can capture optimal position data.
MCCAPTURE_TORQUE	Axis can capture torque data.

CapturePoints Maximum number of data points that may be captured.
CaptureAndCompare High speed position capture and compare:

Value	Description
TRUE	Feature is supported.
FALSE	Feature isn't supported.

HighRate Servo update period, in seconds, for High Speed mode (valid only for servo modules).
MediumRate Servo update period, in seconds, for Medium Speed mode (valid only for servo modules).
LowRate Servo update period, in seconds, for Low Speed mode (valid only for servo modules).
HighStepMin Minimum step rate for High Speed mode (valid only for stepper modules).
HighStepMax Maximum step rate for High Speed mode (valid only for stepper modules).

MediumStepMin Minimum step rate for Medium Speed mode (valid only for stepper modules).
MediumStepMax Maximum step rate for Medium Speed mode (valid only for stepper modules).
LowStepMin Minimum step rate for Low Speed mode (valid only for stepper modules).
LowStepMax Maximum step rate for Low Speed mode (valid only for stepper modules).

Comments

Unlike the other MCAPI structures, the values in this structure are fixed by the hardware configuration and may not be changed.

Before you call **MCGetAxisConfiguration( )** you must set the **cbSize** member to the size of this data structure. C/C++ programmers may use **sizeof( )** , Visual Basic and Delphi programmers will find current sizes for these data structures in the appropriate MCAPI.XXX header file.

Visual Basic users please note that the value used for TRUE in the **MCAXISCONFIG** structure is the Windows standard of 1, not the Basic value of -1. Direct comparisons, such as:

```
   If (Param.CanDoScaling = True) Then
```

will fail. To get correct results use the constant WinTrue, declared in the MCAPI.BAS include file:

```
   If (Param.CanDoScaling = WinTrue) Then
```

Compatibility

There are no compatibility issues with this data structure.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Version: MCAPI 3.0 or higher

See Also
MCGetAxisConfiguration( )

MCCOMMUTATION

MCCOMMUTATION commutation parameters for an *axis*.

```
typedef struct {
   long int cbSize;
   double PhaseA;
   double PhaseB;
   long int Divisor;
   long int PreScale;
   long int Repeat;
} MCCOMMUTATION;
```

Members
cbSize	Size of the **MCCOMMUTATION** data structure, in bytes.
PhaseA	Phase A setting, in degrees.
PhaseB	Phase B setting, in degrees.
Divisor	Commutation divisor.
PreScale	Commutation prescale factor.
Repeat	Commutation repeat count.

Comments
Setting **Divisor**, **PreScale**, or **Repeat** to negative one (-1) will cause **MCSetCommutation( )** to skip setting that value.

Compatibility
The DC2, DCX-PC100, DCX-PCI100, DCX-AT100, and DCX-AT200 controllers do not support a module which is capable of onboard commutation. The MC300, MC302, MC360, and the MC362 modules do not support onboard commutation.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Version: MCAPI 3.2 or higher

See Also
MCSetCommutation( )

MCCONTOUR

MCCONTOUR structure contains contouring parameters for an axis.

```
typedef struct {
   double VectorAccel;
   double VectorDecel;
   double VectorVelocity;
   double VelocityOverride;
} MCCONTOUR;
```

Members

VectorAccel Acceleration value for motion along a contour path.
VectorDecel Deceleration value for motion along a contour path.
VectorVelocity Maximum velocity for motion along a contour path.
VelocityOverride Proportional scaling factor for vector velocity, may be changed while axes are in motion.

Comments

The vector velocity parameter must be set prior to starting a contour path motion and can not be changed once the motion has begun. To change velocity on the fly, set the velocity override to a value other than 1.0. This value is used to proportionally scale the velocities.

Compatibility

The MCAPI does not does not support contouring on the DC2, DCX-PC100, or DCX-PCI100 controllers.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Version: MCAPI 1.0 or higher

See Also

MCGetContourConfig( ), **MCSetContourConfig( )**

MCFILTEREX

MCFILTEREX structure contains the PID filter parameters for a servo axis, or the closed-loop parameters for a stepper axis operating in closed-loop mode. Please see the online MCAPI Reference for the **MCFILTER** structure.

```
typedef struct {
   long int cbSize;
   double Gain;
   double IntegralGain;
   double IntegrationLimit;
   long int IntegralOption;
   double DerivativeGain;
   double DerSamplePeriod;
   double FollowingError;
   double VelocityGain;
   double AccelGain;
   double DecelGain;
   double EncoderScaling;
   long UpdateRate;
} MCFILTEREX;
```

Members

cbSize Size of the **MCFILTEREX** data structure, in bytes.

Gain Proportional Gain setting of the PID loop.

IntegralGain Gain setting for the integral term of the PID loop.

IntegrationLimit Limit value for the integral term, limits the power the integral gain can use to reduce error to zero.

IntegralOption Operating mode for the integral term of the PID loop:

Value	Description
MC_INT_NORMAL	Selects the normal (always on) operation of the integral term.
MC_INT_FREEZE	Freeze the integral term while moving, re-enable after move is complete.
MC_INT_ZERO	Zero and freeze the integral term while moving, re-enable after move is complete.

DerivativeGain Gain setting for the derivative term of the PID loop.

DerSamplePeriod Time interval, in seconds, between derivative samples.

FollowingError Maximum position error, default units are encoder counts.

VelocityGain Gain setting for the feed-forward gain of the PID loop, volts per encoder count per second.

AccelGain Feed-forward acceleration gain setting.

DecelGain Feed-forward deceleration gain setting.

EncoderScaling Encoder counts per step scaling factor for closed-loop steppers (ignored for servos).

UpdateRate This parameter is used to set the feedback loop rate for servo motors and closed-loop steppers, or the maximum stepper pulse rate for open-loop stepper motor axes:

Value	Description
MC_RATE_UNKNOWN	Returned if MCAPI cannot determine the current rate.
MC_RATE_LOW	Selects the normal (always on) operation of the integral term.
MC_RATE_MEDIUM	Freeze the integral term while moving, re-enable after move is complete.
MC_RATE_HIGH	Zero and freeze the integral term while moving, re-enable after move is complete.

Comments

The servo tuning utility program offers a convenient, interactive format for determining appropriate filter settings for your servo/amplifier or closed-loop stepper.

When used with the DCX-PC100 and MC2xx series modules it is not always possible to read the **UpdateRate** parameter from the motion controller (requires recent firmware). If the MCAPI cannot read back this parameter it will return the value MC_RATE_UNKNOWN. If **UpdateRate** is set to MC_RATE_UNKNOWN and a call is made to **MCSetMotionConfigEx( )** the controller's **UpdateRate** value will not be changed.

Compatibility

VelocityGain is not supported on the DCX-PCI100 controller, MC100, MC110 modules, or closed-loop steppers. **AccelGain** is not supported on the DC2, DCX-PC100, or DCX-PCI100 controllers. **DecelGain** is not supported on the DC2, DCX-PC100, or DCX-PCI100 controllers. **EncoderScaling** is not supported on servos. **UpdateRate** is not supported on the DC2 or DCX-PCI100 controllers.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Version: MCAPI 3.2 or higher

See Also

MCGetFilterConfigEx( ), **MCSetFilterConfigEx( )**

MCJOG

MCJOG structure defines jog parameters for an axis.

```
typedef struct {
   double Acceleration;
   double MinVelocity;
   double Deadband;
   double Gain;
   double Offset;
} MCJOG;
```

Members

Acceleration	Acceleration rate for use with jogging.
MinVelocity	Stepper motor jog minimum velocity (this parameter has no effect for servo motors).
Deadband	Deadband specifies a threshold value about the center position of the joystick below which motion of the joystick will not effect motor position. This prevents undesirable drifting of the motor due to mechanical and electrical variations in the joystick.
Gain	Gain value for jogging. This parameter is effectively multiplied by the current joystick position to produce a velocity. To increase the maximum velocity, set **Gain** to a larger value. To reverse the direction of motor travel with respect to joystick direction **Gain** may be set to a negative value.
Offset	Specifies the center position of the joystick, in volts.

Comments
The jog settings determine the performance of an axis when the jogging inputs are active and jogging has been enabled.

Compatibility
The DCX-PCI controllers, DC2 stepper axes, MC150, and MC160 modules do not support jogging.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Version: MCAPI 1.0 or higher

See Also
MCEnableJog( ), **MCGetJogConfig( )**, **MCSetJogConfig( )**

MCMOTIONEX

MCMOTIONEX structure defines basic motion parameters for an axis.

```
typedef struct {
   int cbSize;
   double Acceleration;
   double Deceleration;
   double Velocity;
   double MinVelocity;
   short int Direction;
   double Torque;
   double Deadband;
   double DeadbandDelay;
   short int StepSize;
   short int Current;
   WORD HardLimitMode;
   WORD SoftLimitMode;
   double SoftLimitLow;
   double SoftLimitHigh;
   short int EnableAmpFault;
} MCMOTIONEX;
```

Members

cbSize	Size of the **MCMOTIONEX** data structure, in bytes.
Acceleration	Acceleration rate for motion.
Deceleration	Deceleration rate for motion.
Velocity	Velocity for motion.
MinVelocity	Stepper motor minimum velocity (this parameter has no effect for servo motors).
Direction	Sets the direction of travel for velocity mode operation. Note that the interpretation of positive and negative will depend upon your hardware configuration:

Value	Description
MC_DIR_POSITIVE	Selects the positive travel direction.
MC_DIR_NEGATIVE	Selects the negative travel direction.

Torque Sets the maximum output torque level for servos. When a servo is operated in torque mode this value represents the continuous output level. The default output units are volts, but this may be scaled using the **Constant** member of the **MCSCALE** structure.

Deadband Sets the position dead band value.

DeadbandDelay Time limit that an axis must remain within the dead band area to qualify as "in range". If this value cannot be read back from the controller the Motion Control API function **MCGetMotionConfigEx( )** will set this value to -1. **MCSetMotionConfigEx( )** ignores this parameter if the value is equal to -1.

StepSize Sets the step size output for stepper motor operation:

Value	Description
MC_STEP_FULL	Selects full step operation.
MC_STEP_HALF	Selects half step operation.

Current Selects full or reduced current operation for stepper motors. Reduced current is typically used with stepper motors when they are stopped in a single position for an extended time to reduce motor heating.

Value	Description
MC_CURRENT_FULL	Selects full current (normal) operation.
MCCURRENT_HALF	Selects half current (idle) operation.

HardLimitMode Enables hard (physical) limit switches and selects stopping mode. One or more of the following values may be OR'ed together:

Value	Description
MC_LIMIT_LOW	Enables lower limit.
MC_LIMIT_HIGH	Enables upper limit.
MC_LIMIT_ABRUPT	Selects abrupt stopping mode when a limit is encountered.
MCLIMIT_SMOOTH	Selects smooth stopping mode when a limit is encountered.
MCLIMIT_INVERT	Inverts the polarity of the hardware limit switch inputs. This value may not be used with soft limits.

SoftLimitMode Enables soft (software) limit switches and selects stopping mode. See the description of **HardLimitMode** for details.

SoftLimitLow Sets "position" of low soft limit.

SoftLimitHigh Sets "position" of high soft limit.

EnableAmpFault Controls the amplifier fault input for servo motor axes:

Value	Description
TRUE	Enables amplifier fault input.
FALSE	Disables amplifier fault input.

Comments
All of the basic motion parameters are stored in the **MCMOTIONEX** structure. Many of these parameters also have their own Get/Set functions, to permit setting on the fly.

Compatibility
Acceleration is not supported on the DC2 stepper axes. **Deceleration** is not supported on the DCX-PCI100 controller, DC2 stepper axes, MC100, MC110, MC150, or MC160 modules. **MinVelocity** is not supported on the DCX-PCI100, DCX-PC100, or DC2 controllers. **Torque** is not supported on the DCX-PCI100 controller, MC100, or MC110 modules. **Deadband** is not supported on the DCX-PC100 controller, DC2 stepper axes, MC150, MC160, MC260, MC360, and MC362 modules. **DeadbandDelay** is not supported on the DCX-PC100 controller, DC2 stepper axes, MC150, MC160, MC260, MC360 or MC362 modules. **StepSize** is not supported on the DC2 or DCX-PCI100 controllers. **Current** is not supported on the DC2 or DCX-PCI100 controllers. **SoftLimitMode** is not supported on the DC2 or DCX-PC100 controllers. **SoftLimitLow** is not supported on the DC2 or DCX-PC100 controllers. **SoftLimitHigh** is not supported on the DC2 or DCX-PC100 controllers. **EnableAmpFault** is not supported on the DC2 controllers.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Version: MCAPI 1.0 or higher

See Also
MCGetMotionConfigEx( ), **MCSetMotionConfigEx( )**

MCPARAMEX

MCPARAMEX structure provides basic information about the type and configuration of a controller, including the number of axes and modules supported.

```
typedef struct {
   int cbSize;
   int ID;
   int ControllerType;
   int NumberAxes;
   int MaximumAxes;
   int MaximumModules;
   int Precision;
   int DigitalIO
   int AnalogInput;
   int AnalogOutput;
   int PointStorage;
   int CanDoScaling;
   int CanDoContouring;
   int CanChangeProfile;
   int CanChangeRates;
   int SoftLimits;
   int MultiTasking;
   int AmpFault;
} MCPARAMEX;
```

Members

cbSize Size of the **MCPARAMEX** data structure, in bytes.

ID ID number given this controller during driver setup, permits easy translation of a controller handle back to an ID.

ControllerType OEM controller type identifier. It can be one of the following values:

Value	Description
DCXPC100	DCX series PC100 controller.
DCXAT100	DCX series AT100 controller.
DCXAT200	DCX series AT200 controller.
DC2PC100	DC2 series controller.
DC2STN	DC2 stand-alone series controller.
DCXAT300	DCX series AT300 controller.
DCXPCI300	DCX series PCI300 controller.
DCXPCI100	DCX series PCI100 controller.

NumberAxes Number of axes this controller is currently configured for.

MaximumAxes Maximum number of axes this controller supports.

MaximumModules Maximum number of modules this controller supports.

Precision Best numerical precision of controller:

Value	Description
MC_TYPE_LONG	32 bit integer precision.
MC_TYPE_DOUBLE	64 bit floating point precision.

DigitalIO Contains the number of digital IO channels installed.

AnalogInput The number of installed analog input channels.

AnalogOutput The number of analog output channels.

PointStorage Number of learned points that may be stored using **MCLearnPoint( )**

CanDoScaling Controller support for scaling (see **MCSCALE** structure) flag:

Value	Description
TRUE	Scaling is supported.
FALSE	Scaling isn't supported.

CanDoContouring Controller support for contouring (see **MCCONTOUR** structure) flag:

Value	Description
TRUE	Contouring is supported.
FALSE	Contouring not supported.

CanChangeProfile Controller can change acceleration/deceleration profile::

Value	Description
TRUE	Profile change is supported.
FALSE	Profile change not supported.

CanChangeRates Controller support for selectable rates (see **MCFILTEREX** structure) flag:

Value	Description
TRUE	UpdateRate changing is supported.
FALSE	UpdateRate changing isn't supported.

SoftLimits Controller supports soft limits (see **MCMOTIONEX** structure) flag:

Value	Description
TRUE	Soft Limits are supported.
FALSE	Soft Limits are not supported.

MultiTasking Controller supports multitasking flag:

Value	Description
TRUE	Multitasking is supported.
FALSE	Multitasking is not supported.

AmpFault Controller supports amplifier fault flag:

Value	Description
TRUE	Amplifier fault input is supported.
FALSE	Amplifier fault input is not supported.

Comments

Unlike the other MCAPI structures, the values in this structure are fixed by the hardware configuration and may not be changed. The axis type information that existed in the old **MCPARAM** structure may now be found in the **MCAXISCONFIG** structure.

Before you call **MCGetConfigurationEx( )** you must set the **cbSize** member to the size of this data structure. C/C++ programmers may use **sizeof( )**, Visual Basic and Delphi programmers will find current sizes for these data structures in the appropriate MCAPI.XXX header file.

Visual Basic users please note that the value used for TRUE in the **MCPARAMEX** structure is the Windows standard of 1, not the Basic value of -1. Direct comparisons, such as:

```
   If (Param.CanDoScaling = True) Then
```

will fail. To get correct results use the constant WinTrue, declared in the MCAPI.BAS include file:

```
   If (Param.CanDoScaling = WinTrue) Then
```

Compatibility
There are no compatibility issues with this data structure.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Version: MCAPI 3.0 or higher

See Also
MCGetConfigurationEx( )

MCSCALE

MCSCALE structure defines basic scaling parameters for an axis.

```
typedef struct {
   double Constant;
   double Offset;
   double Rate;
   double Scale;
   double Zero;
   double Time;
} MCSCALE;
```

Members
Constant This factor acts as a scale factor for servo analog outputs. By calibrating your motor/amplifier combination, it is possible to scale the output with **Constant** so that torque settings may be specified directly in ft-lbs.

Offset This offset represents an offset from a servo encoder' index pulse to a zero position.

Rate This factor acts as a multiplier for motion commands time values. The base controller time unit is the second, to convert this to minutes set **Rate** to 60.0, to convert to milliseconds rate should be set to 0.001.

Scale This scaling factor is applied to motion parameters to convert from encoder counts to real world units.

Zero Specifies that a soft zero should be located this distance from actual zero. By moving the soft zero around it is possible to have a series of position

	commands repeated at various spots in the range of travel without modifying the position commands. The actual zero position is not changed by this command.
Time	This is the time factor for controller level wait commands. See the discussion of the **Rate** parameter above for more information on setting this value. Note that a single **Time** value is maintained per controller (i.e. **Time** is axis independent).

Comments

The scale factors provide a consistent, easy method of relating motion values to the actual physical system being controlled.

Compatibility

The DC2, and the DCX-PC100 do not support any of the aforementioned members. The DCX-PCI100 does not support **Offset** or **Constant**.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Version: MCAPI 1.0 or higher

See Also

MCGetScale( ), **MCSetScale( )**

Chapter Contents

Chapter
12

MCAPI Parameter Setup Functions

Parameter setup functions allow the program to consistently configure the motion control card and individual modules to behave in an appropriate manner for a given application. Although trajectory parameters, PID loop gains, and end of travel limits should be set prior to commanding motion, these and other parameters may be changed during a move. However, certain parameters once passed to the card will not alter behavior until **MCEnableAxis( )** is called, which allows the specific axis to then implement several queued parameters at once in a logical and safe fashion. For first time setup, a development tool like **Motion Integrator** should be used to determine the proper tuning parameters that can be passed by the functions in this chapter.

To see examples of how the functions in this chapter are used, please refer to the online Motion Control API Reference.

MCConfigureCompare

MCConfigureCompare( ) configures an axis for high-speed position compare mode operation.

```
long int MCConfigureCompare(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double* values,                // array of compare points
   long int num,                  // number of points in values array
   double inc,                    // increment between equally paced points
   long int mode,                 // output signal mode
   double period                  // output period for one shot mode
                                  // (seconds)
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to configure.
values	Array of compare position values.
num	Number of compare values.
inc	Increment between successive compare positions when in evenly-spaced mode (see Comments, below).

mode Specifies how the controller is to signal that a compare position has been seen:

Value	Description
MC_COMPARE_DISABLE	Disables the output.
MC_COMPARE_INVERT	Inverts active level of the output – may be OR'ed together with any of the other settings for mode.
MC_COMPARE_ONESHOT	Configures the output for one-shot operation. The value for period will be used for the period of the one-shot.
MC_COMPARE_STATIC	Configures the output for static mode (see the controller documentation for details).
MC_COMPARE_TOGGLE	Configures the output to toggle between the active and inactive states each time a compare value is reached.

Returns
This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments
Points for **MCConfigureCompare( )** may be entered in one of two ways. Discrete points, up to the number allowed by the module (typically 512) may be stored in the array *values* and passed to the controller. If the compare points are equally spaced store the beginning point in the first location of *values*, set *num* to one, and set *inc* to the per point increment. Note that *inc* is ignored if it is set equal to or less than zero, or if *num* is set to a value other than one.

The high-speed compare function signals a valid compare by way of a hardware output signal from the motor module. Use the mode flag to configure the operation of this hardware output.

Compatibility
The DC2, DCX-PC100, DCX-AT200, and DCX-PCI100 controllers do not support high-speed position compare.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 3.1 or higher

Prototypes
Delphi: function MCConfigureCompare(hCtlr: HCTRLR; axis: Word; values: Array of Double; num: Longint; inc: Double; mode: LongInt; period: Double): LongInt; stdcall;

VB: Function MCConfigureCompare(ByVal hCtrlr As Integer, ByVal axis As Integer, values As Double, ByVal num As Long, ByVal inc As Double, ByVal mode As Long, ByVal period As Double) As Long

LabVIEW: Not Supported

MCCL Reference
LC, NC, OC, OP

See Also
MCEnableCompare( ), **MCGetCount( )**

MCSetAcceleration

MCSetAcceleration( ) sets programmed acceleration value for the selected axis to *rate*, where *rate* is specified in the current units for *axis*.

```
void MCSetAcceleration(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double rate                    // new acceleration rate
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to change acceleration value of.
rate	New acceleration rate.

Returns

This function does not return a value.

Comments

The acceleration value for a particular *axis* may also be set using the **MCSetMotionConfigEx( )** function; **MCSetAcceleration( )** provides a short-hand method for setting just the acceleration value.

Compatibility

The DC2 stepper axes do not support ramping.

Requirements

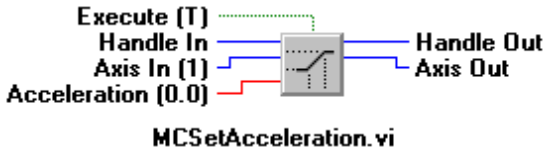
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure MCSetAcceleration(hCtlr: HCTRLR; axis: Word; rate: Double); stdcall;

VB: Sub MCSetAcceleration Lib(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal rate As Double)

LabVIEW:



MCCL Reference

SA

See Also

MCGetAccelerationEx( ), **MCSetMotionConfigEx( )**

MCSetAuxEncPos

MCSetAuxEncPos( ) sets the current position of the auxiliary encoder.

```
void MCSetAuxEncPos(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double position                // new position
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number of auxiliary encoder to set.
position	New encoder position.

Returns

This function does not return a value.

Comments

This command sets the current position of the auxiliary encoder to the value given by the *position* argument. A value of MC_ALL_AXES may be specified for *axis* to set the auxiliary encoders for all axes installed on a controller.

DCX-AT200 firmware version 3.5a or higher, or DCX-PC100 firmware version 4.9a or higher is required if you wish to set the position of the auxiliary encoder to a value other than zero. Earlier firmware versions ignore the value in the Position argument and zero the Auxiliary Encoder.

Compatibility

The DC2, DCX-PCI100 controllers, MC100, MC110, MC150, and MC320 modules do not support auxiliary encoders. Closed-loop steppers do not support auxiliary encoder functions, since the connected encoder is considered a primary encoder.

Requirements

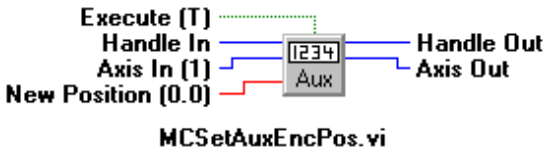
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure MCSetAuxEncPos(hCtlr: HCTRLR; axis: Word; position: Double); stdcall;

VB: Sub MCSetAuxEncPos Lib(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal position As Double)

LabVIEW:



MCCL Reference

AH

See Also

MCGetAuxEncPosEx( )

MCSetCommutation

MCSetCommutation( ) sets the commutation settings for the MC320 module.

```
long int MCSetCommutation(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   MCCOMMUTATION* pCommutation    // pointer to commutation structure
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to which commutation parameters are to be set.
pCommutation	Points to an **MCCOMMUTATION** structure that contains commutation settings for *axis*.

Returns

MCSetCommutation( ) returns the value MCERR_NOERROR if the function completed without errors. If there was an error, one of the MCERR_xxxx error codes is returned.

Comments

See the section on commutation in your DCX-300 Series User's Guide for details on how to set use the commutation features of the MC320 module.

Compatibility

The DC2, DCX-PC100, DCX-PCI100, DCX-AT100, and DCX-AT200 controllers do not support a module which is capable of onboard commutation. The MC300, MC302, MC360, and the MC362 modules do not support onboard commutation.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 3.2 or higher

Prototypes

Delphi:	function MCSetCommutation(hCtlr: HCTRLR; axis: Word; var pCommutation: MCCOMMUTATION): LongInt; stdcall;
VB:	Function MCSetCommutation(ByVal hCtrlr As Integer, ByVal axis As Integer, Commutation As MCCommutation) As Long
LabVIEW:	Not Supported

MCCL Reference

LA, LB, LD, LE, LR

See Also

MCCOMMUTATION structure definition

MCSetContourConfig

MCSetContourConfig( ) sets contouring configuration for the specified axis.

```
short int MCConfigureDigitalIO(
   HCTRLR hCtlr,                 // controller handle
   WORD axis,                    // axis number
   MCCONTOUR* pContour           // address of contouring configuration
                                 // structure
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to set contouring configuration for.
pContour	Points to an **MCCONTOUR** structure that contains contouring configuration information for *axis*.

Returns

The return value is TRUE if the function is successful. A return value of FALSE indicates the function did not find the *axis* specified (*hCtlr* or *axis* incorrect).

Comments

Contouring configuration data should be setup prior to executing any contour motion. The field **CanDoContouring** in the **MCPARAMEX** structure will be set to TRUE, if the controller can process contour configuration data.

Compatibility

The MCAPI does not does not support contouring on the DC2, DCX-PC100, or DCX-PCI100 controllers.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi:	function MCSetContourConfig(hCtlr: HCTRLR; axis: Word; var pContour: MCCONTOUR): SmallInt; stdcall;
VB:	Function MCConfigureDigitalIO(ByVal hCtrlr As Integer, ByVal channel As Integer, ByVal mode As Integer) As Integer
LabVIEW:	Not Supported

MCCL Reference

VA, VD, VO, VV

See Also

MCGetContourConfig( ), **MCCONTOUR** structure definition

MCSetDeceleration

MCSetDeceleration( ) sets programmed deceleration value for the selected axis to *rate*, where *rate* is specified in the current units for *axis*.

```
void MCSetDeceleration(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double rate                    // new deceleration rate
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to change acceleration value of.
rate	New deceleration rate.

Returns

This function does not return a value.

Comments

The deceleration value for a particular *axis* may also be set using the **MCSetMotionConfigEx( )** function; **MCSetDeceleration( )** provides a short-hand method for setting just the deceleration value. A value of MC_ALL_AXES may be specified for *axis* to set the deceleration for all axes installed on a controller.

Compatibility

The DCX-PCI100 controller, MC100, MC110, MC150, and MC160 modules do not support a separate deceleration value. Instead, the acceleration value will also be used as the deceleration value. The DC2 stepper axes do not support ramping.

Requirements

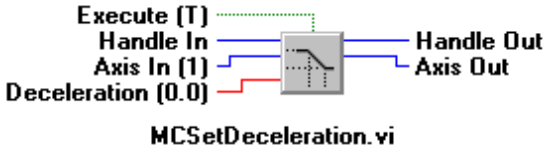
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure MCSetDeceleration(hCtlr: HCTRLR; axis: Word; rate: Double); stdcall;

VB: Sub MCSetDeceleration(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal rate As Double)

LabVIEW:



MCCL Reference

DS

See Also

MCGetDecelerationEx( ) , **MCSetMotionConfigEx( )**

MCSetDigitalFilter

MCSetDigitalFilter( ) sets the digital filter coefficients for the specified axis.

```
long int MCSetDigitalFilter(
   HCTRLR hCtlr,                 // controller handle
   WORD axis,                    // axis number
   double* pCoeff,               // array of digital filter coefficients
   long int num                  // number of coefficients
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number.
pCoeff	Array of coefficients, must be *num* elements long (or longer). If the pointer is NULL the filter will be zeroed (overwriting any previous settings) but no new filter values will be stored.
num	Number of coefficients to retrieve, cannot be larger than the maximum digital filter size supported by the controller.

Returns

This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments

This sets zero or more of the digital filter coefficients for the specified axis. The number of coefficients cannot exceed the maximum value supported by the axis, as reported by **MCGetCount( )**. Calling **MCSetDigitalFilter( )** overwrites any filter values previously downloaded to this axis.

Compatibility

The DC2, DCX-PC100, DCX-AT200, DCX-PCI100 controllers, MC360, and MC362 modules do not support digital filtering.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 3.1 or higher

Prototypes

Delphi:	function MCSetDigitalFilter(hCtlr: HCTRLR; axis: Word; pCoeff: Array of Double; num: Longint):Longint; stdcall;
VB:	Function MCSetDigitalFilter(ByVal hCtrlr As Integer, ByVal axis As Integer, coeff As Double, ByVal num As Integer) As Long
LabVIEW:	Not Supported

MCCL Reference

FL, ZF

See Also

MCEnableDigitalFilter( ) , **MCGetCount( )**, **MCGetDigitalFilter( )**, **MCIsDigitalFilter( )**

MCSetFilterConfigEx

MCSetFilterConfigEx( ) configures the PID loop settings for a servo motor or the closed-loop settings for a stepper motor operating in closed-loop mode. Please see the online MCAPI Reference for the **MCSetFilterConfig( )** prototype.

```
long int MCSetFilterConfigEx(
   HCTRLR hCtlr,                 // controller handle
   WORD axis,                    // axis number
   MCFILTEREX* pFilter           // pointer to PID filter structure
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number from which to retrieve PID information.
pFilter	Points to a **MCFILTEREX** structure that contains PID filter configuration information for *axis*.

Returns

MCSetFilterConfigEx( ) returns the value MCERR_NOERROR if the function completed without errors. If there was an error, one of the MCERR_xxxx error codes is returned.

Comments

The easiest way to change filter settings is to first call **MCGetFilterConfigEx( )** to obtain the current PID filter settings for *axis*, modify the values in the **MCFILTEREX** structure, and write the changed settings back to *axis* with **MCSetFilterConfigEx( )**.

Closed-loop stepper operation requires firmware version 2.1a or higher on the DCX-PCI300 and firmware version 2.5a or higher on the DCX-AT300.

Compatibility

VelocityGain is not supported on the DCX-PCI100 controller, MC100, MC110 modules, or closed-loop steppers. **AccelGain** is not supported on the DC2, DCX-PC100, or DCX-PCI100 controllers. **DecelGain** is not supported on the DC2, DCX-PC100, or DCX-PCI100 controllers.

Requirements

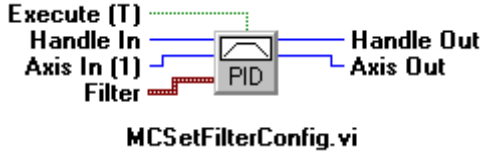
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 3.2 or higher

Prototypes

Delphi: function MCSetFilterConfigEx(hCtlr: HCTRLR; axis: Word; var pFilter: MCFILTEREX): SmallInt; stdcall;

VB: Function MCSetFilterConfigEx(ByVal hCtrlr As Integer, ByVal axis As Integer, filter As MCFilterEx) As Integer

LabVIEW:



MCCL Reference
AG, DG, FR, IL, SD, SE, SI, VG

See Also
MCGetFilterConfigEx( ), **MCFILTEREX** structure definition

MCSetGain

MCSetGain( ) sets the proportional gain of a servo's feedback loop.

```
long int MCSetGain(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double gain                    // new gain setting
);
```

Parameters
hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to change gain of.
gain	New proportional gain.

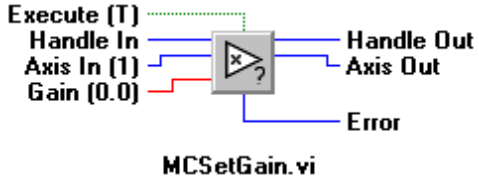
Returns
This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments
The gain value for a particular *axis* may also be set using the **MCSetMotionConfigEx( )** function; **MCSetGain( )** provides a short-hand method for setting just the gain value and for updating gain settings on the fly when operating in gain mode.

Compatibility
The MCAPI does not support closed-loop functionality on any stepper axes at this time.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes
Delphi: function MCSetGain(hCtlr: HCTRLR; axis: Word; gain: Double): Longint; stdcall;
VB: Function MCSetGain(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal gain As Double) As Long
LabVIEW:



MCCL Reference
SG

See Also
MCGetGain( ), **MCSetMotionConfigEx( )**

MCSetJogConfig

MCSetJogConfig( ) sets jog configuration for the specified axis.

```
short int MCSetJogConfig(
   HCTRLR hCtlr,                // controller handle
   WORD axis,                   // axis number
   MCJOG* pJog                  // address of jog configuration structure
);
```

Parameters
hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to configure jog information.
pJog	Points to a **MCJOG** structure that contains jog configuration information for *axis*.

Returns
The return value is TRUE if the function is successful. Otherwise it returns FALSE, indicating the function did not find the *axis* specified (hCtlr or *axis* incorrect).

Comments
It is important to set the jog configuration before enabling jogging if you will be using non-default parameters for the jog configuration.

Compatibility
The DCX-PCI controllers, DC2 stepper axes, MC150, and MC160 modules do not support jogging.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes
Delphi:	function MCSetJogConfig(hCtlr: HCTRLR; axis: Word; var pJog: MCJOG): SmallInt; stdcall;
VB:	Function MCSetJogConfig(ByVal hCtrlr As Integer, ByVal axis As Integer, jog As MCJog) As Integer
LabVIEW:	Not Supported

MCCL Reference
JA, JB, JG, JO, JV

See Also

MCEnableJog( ), **MCGetJogConfig( )**, **MCJOG** structure definition

MCSetLimits

MCSetLimits( ) sets the current hard and soft limit settings for the specified axis.

```
long int MCSetLimits(
   HCTRLR hCtlr,                // controller handle
   WORD axis,                   // axis number
   short int hardMode,          // hard limit mode flags
   short int softMode,          // soft limit mode flags
   double limitMinus,           // soft negative limit value
   double limitPlus             // soft positive limit value
);
```

Parameters

hCtlr Controller handle, returned by a successful call to **MCOpen( )**.

axis Axis number to set the limits of .

hardMode Combination of the following limit mode flags for the hard limits:

Value	Description
MC_LIMIT_PLUS	Enables the positive limit.
MC_LIMIT_MINUS	Enables the negative limit.
MC_LIMIT_BOTH	Enables both the positive and negative limits.
MC_LIMIT_OFF	Sets the limit stopping mode to turn the motor off when a limit is tripped.
MC_LIMIT_ABRUPT	Sets the limit stopping mode to abrupt (target position is set to current position and PID loop stops axis as quickly as possible).
MC_LIMIT_SMOOTH	Sets the limit stopping mode to smooth (axis executes pre-programmed deceleration when limit is tripped).
MC_LIMIT_INVERT	Inverts the polarity of the hardware limit switch inputs. This value may not be used with soft limits.

softMode Combination of limit mode flags for the soft limits. See the values for *hardMode*, above.

limitMinus Positive limit value for soft limits, if supported by this controller.

limitPlus Negative limit value for soft limits, if supported by this controller.

Returns

MCSetLimits( ) returns the value MCERR_NOERROR if the function completed without errors. If there was an error, one of the MCERR_xxxx error codes is returned, and the limit settings will be left in an undetermined state.

Comments

The limit settings are the same as those that may be set by the **MCSetMotionConfigEx( )** function, however, this function provides a short-hand method for setting just the limit settings.

To disable limits (hard or soft) set the corresponding limit mode variable (*hardMode* and *softMode*) to zero (0). To disable a particular limit (plus or minus) DO NOT include its corresponding mode flag (MC_LIMIT_PLUS or MC_LIMIT_MINUS, respectively) in the combination of flags that make up the *hardMode* and *softMode* values.

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility

The DC2 and DCX-PC100 controllers do not support soft limits.

Requirements

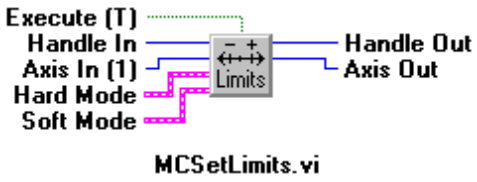
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes

Delphi: function MCSetLimits(hCtlr: HCTRLR; axis: Word; hardMode, softMode: SmallInt; limitMinus, limitPlus: Double): Longint; stdcall;

VB: Function MCSetLimits(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal hardMode As Integer, ByVal SoftMode As Integer, ByVal limitMinus As Double, ByVal limitPlus As Double) As Long

LabVIEW:



MCCL Reference

HL, LF, LL, LM, LN

See Also

MCGetMotionConfigEx( ), **MCGetLimits( )**, **MCSetMotionConfigEx( )**

MCSetModuleInputMode

MCSetModuleInputMode( ) sets the current input mode for the specified axis.

```
long int MCSetModuleInputMode(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double mode                    // input mode value
);
```

Parameters

hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number of which to set input mode.
mode Input mode for the specified axis:

Value	Description
MC_IM_OPENLOOP	Sets stepper motor axis to open-loop mode.
MC_IM_CLOSEDLOOP	Sets stepper motor axis to closed-loop mode.

Returns

The return value is MCERR_NOERROR if no errors were detected. If there was an error, one of the MCERR_xxxx error codes is returned and the variable pointed to by *mode* is left unchanged.

Comments

You will need to issue **MCEnableAxis( )** twice, once FALSE and once TRUE, after calling this function to assure proper changing of modes.

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility

The DC2, DCX-PC100, DCX-PCI100, DCX-AT100, and DCX-AT200 controllers do not support a module which is capable of closed-loop stepper operation. The MC362 module is not capable of closed-loop stepper operation.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 3.2 or higher

Prototypes

Delphi: function MCSetModuleInputMode(hCtlr: HCTRLR; axis, mode: LongInt): LongInt; stdcall;
VB: Function MCSetModuleInputMode(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal mode As Long) As Long
LabVIEW: Not Supported

MCCL Reference

IM

See Also

MCGetModuleInputMode( )

MCSetModuleOutputMode

MCSetModuleOutputMode( ) configures the output of the specified servo or stepper axis.

```
void MCSetModuleOutputMode(
   HCTRLR hCtlr,                // controller handle
   WORD axis,                   // axis number
   double mode                  // output mode selection
);
```

Parameters

hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number to set output mode of.
mode Output mode, one of the following constants:

Value	Description
MC_OM_BIPOLAR	Sets servo axis to bipolar operation. (-10V to +10V)
MC_OM_UNIPOLAR	Sets servo axis to unipolar operation. (0V to +10V, with a separate direction signal)
MC_OM_PULSE_DIR	Sets stepper axis to pulse and direction output.
MC_OM_CW_CCW	Sets stepper axis to clockwise and counter-clockwise operation.

Returns

This function does not return a value.

Comments

Note that the function arguments will depend upon the type of axis being addressed - stepper or servo. Output phase settings are normally made at power up (before motors are energized) and then left unchanged. Incorrect settings can lead to unpredictable operation.

Compatibility

The DC2, DCX-PC100, DCX-PCI100 controllers, MC100, MC110, MC150, and MC160 modules do not support changing the output mode.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure MCSetModuleOutputMode(hCtlr: HCTRLR; axis, mode: Word); stdcall;
VB: Sub MCSetModuleOutputMode(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal mode As Integer)
LabVIEW: Not Supported

MCCL Reference

OM

See Also

MCGetServoOutputPhase( )

MCSetMotionConfigEx

MCSetMotionConfigEx( ) configures an axis for motion.

```
short int MCSetMotionConfigEx(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   MCMOTIONEX* pMotion            // address of motion configuration
                                  // structure
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to configure.
pMotion	Points to a **MCMOTIONEX** structure that contains motion configuration information for the specified axis.

Returns

The return value is TRUE if the function is successful. A return value of FALSE indicates the function could not configure the axis.

Comments

This function provides a way of setting all motion parameters for a given *axis* with a single function call using an initialized **MCMOTIONEX** structure. When you need to setup many of the parameters for an *axis* it is easier to call **MCGetMotionConfigEx( )**, update the **MCMOTIONEX** structure, and write the changes back using **MCSetMotionConfigEx( )**, rather than use a Get/Set function call for each parameter.

Note that some less often used parameters will only be accessible from this function and from **MCGetMotionConfigEx( )** - they do not have individual Get/Set functions.

Compatibility

Acceleration is not supported on the DC2 stepper axes. **Deceleration** is not supported on the DCX-PCI100 controller, MC100, MC110, MC150, or MC160 modules. **MinVelocity** is not supported on the DCX-PCI100, DCX-PC100, or DC2 controllers. **Torque** is not supported on the DCX-PCI100 controller, MC100, or MC110 modules. **Deadband** is not supported on the DCX-PC100 controller, DC2 stepper axes, MC150, MC160, MC260, MC360, or MC362 modules. **DeadbandDelay** is not supported on the DCX-PC100 controller, DC2 stepper axes, MC150, MC160, MC260, MC360 or MC362 modules. **StepSize** is not supported on the DC2 or DCX-PCI100 controllers. **Current** is not supported on the DC2 or DCX-PCI100 controllers. **SoftLimitMode** is not supported on the DC2 or DCX-PC100 controllers. **SoftLimitLow** is not supported on the DC2 or DCX-PC100 controllers. **SoftLimitHigh** is not supported on the DC2 or DCX-PC100 controllers. **EnableAmpFault** is not supported on the DC2 controllers. **UpdateRate** is not supported on the DC2 or DCX-PCI100 controllers.

Requirements

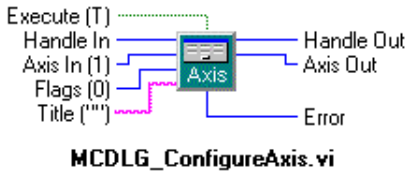
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: function MCSetMotionConfigEx(hCtlr: HCTRLR; axis: Word; var pMotion: MCMOTIONEX): SmallInt; stdcall;

VB: Function MCSetMotionConfigEx(ByVal hCtrlr As Integer, ByVal axis As Integer, motion As MCMotionEx) As Integer

LabVIEW:



MCCL Reference

DB, DI, DT, FC, FF, FN, FR, HC, HS, LM, LS, MS, MV, SA, SD, SF, SG, SH, SI, SQ, SV

See Also

MCGetMotionConfigEx( ), **MCMOTIONEX** structure definition

MCSetOperatingMode

MCSetOperatingMode( ) sets the controller operating mode for *axis*.

```
void MCSetOperatingMode(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   WORD master,                   // master contouring axis
   WORD mode                      // new operating mode
);
```

Parameters

hCtlr Controller handle, returned by a successful call to **MCOpen( )**.

axis Axis number to configure.

master Contouring master axis (used for contour mode only).

mode New operating mode, can be any of the following:

Value	Description
MC_MODE_CONTOUR	Selects contouring mode (must also specify *master*).
MC_MODE_GAIN	Selects gain mode of operation.
MC_MODE_POSITION	Selects the position mode of operation (default).
MC_MODE_TORQUE	Selects torque mode operation.
MC_MODE_VELOCITY	Selects the velocity mode.

Returns

This function does not return a value.

Comments

This function is used to switch between the main operating modes of the controller. All modes except MC_MODE_CONTOUR are supported by all controllers. Programs can check the field **CanDoContouring** of the **MCPARAMEX** structure for the value TRUE to determine if a controller can operate in MC_MODE_CONTOUR mode.

This function should not be called while *axis* is in motion.

Compatibility

The MCAPI does not does not support contouring on the DC2, DCX-PC100, or DCX-PCI100 controllers. Gain mode is not supported on stepper axes, MC100, or MC110 modules. Torque mode is not supported on stepper axes, DCX-PCI100 controller, MC100, or MC110 modules.

Requirements

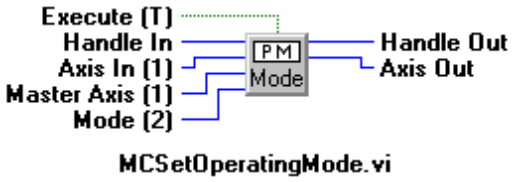
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure MCSetOperatingMode(hCtlr: HCTRLR; axis, master, mode: Word); stdcall;

VB: Sub MCSetOperatingMode(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal master As Integer, ByVal mode As Integer)

LabVIEW:



MCCL Reference

CM, GM, PM, QM, VM

See Also

Controller hardware manual

MCSetPosition

MCSetPosition( ) sets the current position for *axis* to *position*.

```
void MCSetPosition(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double position                // new position
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to change position of.
position	New position value.

Returns

This function does not return a value.

Comments

The current position of *axis* will be immediately updated to the value of *position*.

This function may be called with *axis* set to MC_ALL_AXES set the position of all axes at once. All axes will be set to the same value of *position*.

Compatibility

There are no compatibility issues with this function.

Requirements

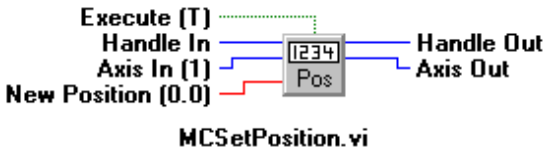
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure MCSetPosition(hCtlr: HCTRLR; axis: Word; position: Double); stdcall;

VB: Sub MCSetPosition(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal position As Double)

LabVIEW:



MCCL Reference

DH

See Also

MCGetPositionEx( )

MCSetRegister

MCSetRegister( ) sets the value of the specified general purpose register.

```
long int MCSetRegister(
   HCTRLR hCtlr,                // controller handle
   long int register,           // register number
   void* pValue,                // pointer to variable with new register
                                // value
   long int type                // type of variable pointed to by pValue
);
```

Parameters

hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
register Register number to read from (0 to 255).
pValue Pointer to a variable that will has the new value for the register.
type Type of data pointed to by *pValue*:

Value	Description
MC_TYPE_LONG	Indicates *pValue* points to a variable of type long integer.
MC_TYPE_DOUBLE	Indicates *pValue* points to a variable of type double precision floating point.
MC_TYPE_FLOAT	Indicates *pValue* points to a variable of type single precision floating point.

Returns

The return value is MCERR_NOERROR, if no errors were detected. However, if there was an error, the return value is one of the MCERR_xxxx error codes, and the register value is unpredictable.

Comments

MCSetRegister( ) and **MCGetRegister( )** allow you to write to and read from, respectively, the general purpose registers on the motion controller. When running background tasks on a multitasking controller the only way to communicate with the background tasks is to pass parameters in the general purpose registers.

You cannot write to the local registers (registers 0 - 9) of a background task. When you need to communicate with a background task be sure to use one or more of the global registers (10 - 255).

To determine if your controller supports multi-tasking check the **MultiTasking** field of the **MCPARAMEX** structure returned by **MCGetConfigurationEx( )**.

Compatibility

There are no compatibility issues with this function.

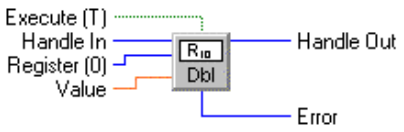
Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 2.0 or higher

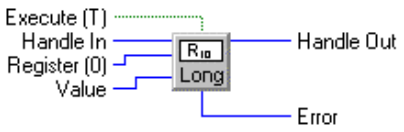
Prototypes

Delphi: function MCSetRegister(hCtlr: HCTRLR; register: Longint; var pValue: Pointer; type: Longint): Longint; stdcall;
VB: Function MCSetRegister(ByVal hCtrlr As Integer, ByVal register As Long, value As Any, ByVal argtype As Long) As Long

LabVIEW:



MCSetRegisterDouble.vi



MCSetRegisterLong.vi

MCCL Reference

AL, AR

See Also

MCGetRegister( )

MCSetScale

MCSetScale( ) sets scaling for the specified axis to the values contained in the **MCSCALE** structure.

```
short int MCSetScale(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   MCSCALE* pScale                // updated scaling settings
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to change scale of.
pScale	Pointer to structure with new scale values.

Returns

This function returns TRUE, if the functions completes successfully. A return value of FALSE indicates there was an error (*hCtlr* or *axis* is invalid).

Comments

Setting scaling factors allows application programs to talk to the controller in real world units, as opposed to arbitrary "encoder counts". You can determine if a controller can process scaling requests by testing the **CanDoScaling** flag in the **MCPARAMEX** structure for the controller.

This function may be called with *axis* set to MC_ALL_AXES to set the scaling of all axes at once. All axes will be set to the same value.

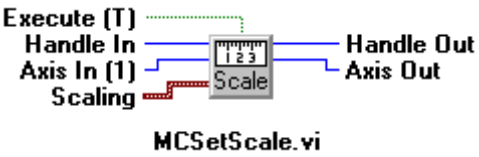
When **Scale** to a value other than one, **SoftLimitLow** and **SoftLimitHigh** should be changed to accommodate the new real world units.

Compatibility

The DC2 and the DCX-PC100 do not support any scaling members. The DCX-PCI100 does not support **Offset** or **Constant**.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes
Delphi: function MCSetScale(hCtlr: HCTRLR; axis: Word; var pScale: MCSCALE): SmallInt; stdcall;
VB: Function MCSetScale(ByVal hCtrlr As Integer, ByVal axis As Integer, scale As MCScale) As Integer
LabVIEW:



MCCL Reference
UK, UO, UR, US, UT, UZ

See Also
MCGetConfigurationEx( ), **MCGetScale( )**, **MCPARAMEX** structure definition

MCSetServoOutputPhase

MCSetServoOutputPhase( ) sets the output phasing for the specified servo *axis*.

```
void MCSetServoOutputPhase(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   WORD phase                     // desired phasing
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number to change servo phase of.
phase Desired phasing, one of the following:

Value	Description
MC_PHASE_STD	Selects standard or normal phasing. (default)
MC_PHASE_REV	Selects reverse phasing.

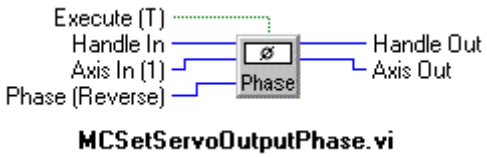
Returns
This function does not return a value.

Comments
This function may be called with *axis* set to MC_ALL_AXES set the phase of all axes at once. All axes will be set to the same value of *phase*.

Compatibility
The MC100 and MC110 modules do not support phase reverse.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes
Delphi: procedure MCSetServoOutputPhase(hCtlr: HCTRLR; axis, phase: Word); stdcall;
VB: Sub MCSetServoOutputPhase(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal mode As Integer)
LabVIEW:



MCCL Reference
PH

See Also
MCGetServoOutputPhase( )

MCSetTorque

MCSetTorque( ) sets maximum output level for servos.

```
long int MCSetTorque(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double torque                  // new torque setting
);
```

Parameters
hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to change torque of.
torque	New torque.

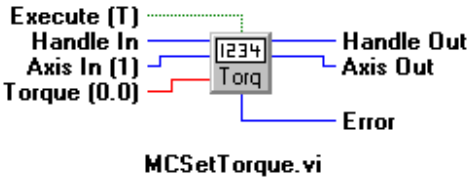
Returns
This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments
The torque value for a particular *axis* may also be set using the **MCSetMotionConfigEx( )** function; **MCSetTorque( )** provides a short-hand method for setting just the torque value and for updating torque settings on the fly when operating in torque mode.

Compatibility
Torque mode is not supported on stepper axes, DCX-PCI100 controller, MC100, or MC110 modules.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes
Delphi: function MCSetTorque(hCtlr: HCTRLR; axis: Word; torque: Double): Longint; stdcall;
VB: Not Supported
LabVIEW:



MCCL Reference
SQ

See Also
MCGetTorque( ), **MCSetMotionConfigEx( )**

MCSetVectorVelocity

MCSetVectorVelocity( ) sets the vector velocity for the specified axis, in whatever units the axis is configured for.

```
long int MCSetVectorVelocity(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double velocity                // new vector velocity value
);
```

Parameters
hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to set vector velocity of.
velocity	New vector velocity value for the specified axis.

Returns
This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments
The vector velocity value for a particular *axis* may also be set using **MCSetContourConfig( )**; **MCSetVectorVelocity( )** provides a short-hand method for setting just the vector velocity value and is most useful when updating vector velocity settings on the fly.

Compatibility
The MCAPI does not does not support contouring on the DC2, DCX-PC100, or DCX-PCI100 controllers.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 2.0 or higher

Prototypes
Delphi: function MCSetVectorVelocity(hCtlr: HCTRLR; axis: Word; velocity: Double): Longint; stdcall;
VB: Function MCSetVectorVelocity(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal velocity As Double) As Long
LabVIEW: Not Supported

MCCL Reference
VV

See Also
MCGetVectorVelocity( ), **MCSetContourConfig( )**

MCSetVelocity

MCSetVelocity( ) sets programmed velocity for the selected *axis* to *rate*, where *rate* is specified in the current units for *axis*.

```
void MCSetVelocity(
   HCTRLR hCtlr,                // controller handle
   WORD axis,                   // axis number
   double rate                  // new velocity
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number to change velocity of.
rate New velocity.

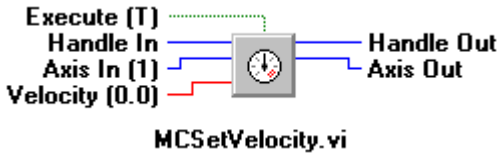
Returns
This function does not return a value.

Comments
The velocity value for a particular *axis* may also be set using the **MCSetMotionConfigEx( )** function; **MCSetVelocity( )** provides a short-hand method for setting just the velocity value and for updating velocity settings on the fly when operating in velocity mode.

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes
Delphi: procedure MCSetVelocity(hCtlr: HCTRLR; axis: Word; rate: Double); stdcall;
VB: Sub MCSetVelocity Lib(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal rate As Double)
LabVIEW:



MCCL Reference
SV

See Also
MCGetVelocityEx( ), **MCSetMotionConfigEx( )**

Chapter Contents

Chapter
13

MCAPI Motion Functions

Motion functions range in use from allowing the program to commence or cease motion to permitting control of sequencing to altering operation of axes during motion.

A word of caution must be given regarding the use of board-level sequencing commands. Even though each of these functions includes a warning in this chapter, it should be stressed that once a command containing the word “Wait” or “Find” in the command name is called, the board will not accept another command nor will it respond to the calling program until the board has completed what it was initially told to do. This can lead to scenarios where the calling program has absolutely no control during potentially dangerous or otherwise expensive situations.

To see examples of how the functions in this chapter are used, please refer to the online Motion Control API Reference.

MCAbort

MCAbort( ) aborts any current motion for the specified axis or axes.

```
void MCAbort(
   HCTRLR hCtlr,                // controller handle
   WORD axis                    // axis number
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to abort motion.

Returns

This function does not return a value.

Comments

The selected *axis* will execute an emergency stop following this command. Issuing this command with *axis* set to MC_ALL_AXES will abort motion for all axes installed on the motion controller.

Servo axes will stop abruptly, and the servo control loop will remain energized.

For stepper motors, pulses from the motion controller will be disabled immediately. The state of the axis (enabled or disabled) following the call to **MCAbort( )** will depend upon the type of controller (see your controller hardware manual).

Following a call to **MCAbort( )**, verify that the axis has stopped using **MCIsStopped( )** or **MCWaitForStop( )**. Then call **MCEnableAxis( )** prior to issuing another motion command.

Following a call to **MCAbort( )** on the DCX-PC100 controller when in velocity mode, call **MCSetOperatingMode( )** prior to issuing another motion command.

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes
Delphi: procedure MCAbort(hCtlr: HCTRLR; axis: Word); stdcall;
VB: Sub MCAbort(ByVal hCtrlr As Integer, ByVal axis As Integer)
LabVIEW:

MCCL Reference
AB

See Also
MCEnableAxis( ), **MCSetOperatingMode( )**, **MCStop( )**, **MCIsStopped( )**, **MCWaitForStop( )**

MCArcCenter

MCArcCenter( ) specifies the center of an arc for contour path motion.

```
long int MCArcCenter(
   HCTRLR hCtlr,                // controller handle
   WORD axis,                   // axis number
   long int type,               // absolute or relative
   double position              // center position
);
```

Parameters

hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number to specify arc center for.
type Flag to indicate if the center position is specified in absolute units or relative to the current position.

Value	Description
MC_ABSOLUTE	Center position is specified in absolute units.
MC_RELATIVE	Center position is specified relative to the current position of *axis*.

position Absolute or relative arc center position for *axis*.

Returns

This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments

This function sets the center of an arc for contour path motion. Since arc motion is performed by two axes, this function should be called twice in a contour path block, once for each axis. To determine if a particular controller can process the **MCArcCenter( )** contouring function, check the **CanDoContouring** flag of the **MCPARAMEX** structure.

Compatibility

The MCAPI does not does not support contouring on the DC2, DCX-PC100, or DCX-PCI100 controllers.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 2.0 or higher

Prototypes

Delphi: function MCArcCenter(hCtlr: HCTRLR; axis: Word; type: SmallInt; position: Double): Longint; stdcall;
VB: Function MCArcCenter (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal arctype As Integer, ByVal position As Double) As Long
LabVIEW: Not Supported

MCCL Reference

CA, CR

See Also

MCArcEndAngle( ), **MCArcRadius( )**, **MCBlockBegin( )**, **MCSetOperatingMode( )**

MCArcEndAngle

MCArcEndAngle( ) specifies the ending angle of an arc for contour path motion.

```
long int MCArcEndAngle(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   long int type,                 // absolute or relative
   double angle                   // ending angle
);
```

Parameters

hCtlr Controller handle, returned by a successful call to **MCOpen( )**.

axis Axis number to specify arc ending angle for.

type Flag to indicate if the end angle is specified in absolute units or relative to the current position.

Value	Description
MC_ABSOLUTE	Center position is specified in absolute units.
MC_RELATIVE	Center position is specified relative to the current position of *axis*.

angle Absolute or relative arc ending angle for *axis*.

Returns

This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments

This function sets the ending angle of an arc for contour path motion function should be called twice in a contour path block, once for each axis. To determine if a particular controller can process the **MCArcCenter( )** contouring function, check the **CanDoContouring** flag of the **MCPARAMEX** structure.

Compatibility

The MCAPI does not does not support contouring on the DC2, DCX-PC100, or DCX-PCI100 controllers.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 2.2 or higher

Prototypes

Delphi: function MCArcEndAngle(hCtlr: HCTRLR; axis: Word; type: SmallInt; angle: Double): Longint; stdcall;

VB: Function MCArcEndAngle (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal arctype As Integer, ByVal angle As Double) As Long

LabVIEW: Not Supported

MCCL Reference

EA, ER

See Also

MCArcCenter( ), **MCArcRadius( )**, **MCBlockBegin( )**, **MCSetOperatingMode( )**

MCArcRadius

MCArcRadius( ) specifies the radius of an arc for contour path motion.

```
long int MCArcRadius(
   HCTRLR hCtlr,                // controller handle
   WORD axis,                   // axis number
   double radius                // arc radius
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to specify arc radius for.
radius	Arc radius for *axis*.

Returns

This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments

This function sets the radius of an arc for contour path motion. To determine if a particular controller can process the **MCArcCenter( )** contouring function, check the **CanDoContouring** flag of the **MCPARAMEX** structure.

Compatibility

The MCAPI does not does not support contouring on the DC2, DCX-PC100, or DCX-PCI100 controllers.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 2.2 or higher

Prototypes

Delphi:	function MCArcRadius(hCtlr: HCTRLR; axis: Word; radius: Double): Longint; stdcall;
VB:	Function MCArcRadius(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal radius As Double) As Long
LabVIEW:	Not Supported

MCCL Reference

RR

See Also

MCArcCenter( ), **MCArcEndAngle( )**, **MCBlockBegin( )**, **MCSetOperatingMode( )**

MCCaptureData

MCCaptureData( ) configures a controller to perform data capture for the specified axis. Captured data includes actual position vs. time, optimal position vs. time, and following error vs. time.

```
long int MCCaptureData(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   long int points,               // number of data points to collect
   double period,                 // time period between data points
                                  // (seconds)
   double delay                   // delay prior to data capture (seconds)
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to capture data.
points	Number of data points to collect.
period	Time period between subsequent data point captures.
delay	Delay (dwell) before initial data collection.

Returns

This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments

Captured position data is typically used to analyze servo motor performance and PID loop tuning parameters. PMC's Servo Tuning utility uses this function to analyze servo performance.

MCBlockBegin( ) may be used with **MCCaptureData( )** to bundle the capture data command with mode and move commands (see the example below).

Beginning with version 3.0 of the MCAPI users may use the **MCGetAxisConfiguration( )** function to determine the data capture capabilities of an axis.

Compatibility

The DC2 stepper axes, and the MC100, MC110, MC150, MC160 modules when installed on the DCX-PC100 controller do not support data capture. The DCX-PCI100 controller does not support torque mode nor do any stepper axes, which prevents the capture of torque values.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes

Delphi:	function MCCaptureData(hCtlr: HCTRLR; axis: Word; points: Longint; period, delay: Double): Longint; stdcall;
VB:	Function MCCaptureData(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal points As Long, ByVal period As Double, ByVal delay As Double) As Long
LabVIEW:	Not Supported

MCCL Reference

PR

See Also

MCGetConfigurationEx( ), **MCGetCaptureData( )**, **MCBlockBegin( )**

MCContourDistance

MCContourDistance( ) sets the distance for user defined contour path motions.

```
long int MCContourDistance(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double distance                // path distance
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number of controlling axis for contour motion.
distance	Path distance for user path.

Returns

This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments

This function is used to specify the distance, as measured along the path, from the contour path starting point to the end of the next motion. It is required for user defined contour path motions.

Compatibility

The MCAPI does not does not support contouring on the DC2, DCX-PC100, or DCX-PCI100 controllers.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 2.0 or higher

Prototypes

Delphi:	function MCContourDistance(hCtlr: HCTRLR; axis: Word; distance: Double): Longint; stdcall;
VB:	Function MCContourDistance(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal distance As Double) As Long
LabVIEW:	Not Supported

MCCL Reference

CD

See Also

MCBlockBegin( )

MCDirection

MCDirection( ) sets the direction of motion when operating in velocity mode.

```
void MCDirection(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double dir                     // new direction
);
```

Parameters

hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number to set the direction of.
dir New direction to move in, may be either of the following values:

Value	Description
MC_DIR_POSITIVE	Selects the positive direction for motion.
MC_DIR_NEGATIVE	Selects the negative direction for motion.

Returns

This function does not return a value.

Comments

This command may be used to change the direction of travel when an axis is operating in Velocity Mode. The actual direction of travel for MC_DIR_POSITIVE and MC_DIR_NEGATIVE will depend upon your hardware configuration.

Compatibility

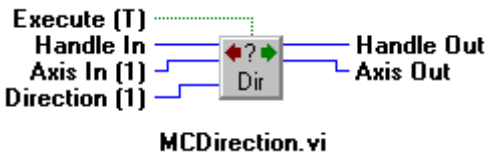
There are no compatibility issues with this function.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure MCDirection(hCtlr: HCTRLR; axis, dir: Word); stdcall;
VB: Sub MCDirection(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal dir As Integer)
LabVIEW:



MCCL Reference

DI

See Also

MCSetOperatingMode( )

MCEdgeArm

MCEdgeArm( ) arms the edge capture function of an open-loop stepper axis.

```
long int MCEdgeArm(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double position                // new position for edge
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number for which to search for the home input signal.
position	The position where the home input signal is sensed for the axis will be properly set to *position* only after a call to **MCWaitForEdge( )** and **MCEnableAxis( )**.

Returns

This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments

This function is used to initialize a stepper motor at a given position. The function remains pending until the home input of the module goes active. At that time you must call **MCWaitForEdge( )** followed by **MCEnableAxis( )** so that the position where the home signal is sensed will be set to the value of the *position* parameter. This function does not cause any motion to be started or stopped.

For the position where the home input signal is sensed to be set to the value of the *position* parameter, you must call **MCWaitForEdge( )** followed by **MCEnableAxis( )**. **MCIsEdgeFound( )** should be used to assure that the home input has latched prior to calling **MCWaitForEdge( )**.

Compatibility

This function is not supported by the DCX-AT200, DCX-PC, or DC2 controllers. The MC300 and MC360 module when in closed-loop mode do not support this function.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 3.2 or higher

Prototypes

Delphi:	function MCEdgeArm(hCtlr: HCTRLR; axis: Word; position: Double): Longint; stdcall;
VB:	Function MCEdgeArm(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal position As Double) As Long
LabVIEW:	Not Supported

MCCL Reference

EL

See Also
MCFindEdge( ), **MCIsEdgeFound( )**, **MCWaitForEdge( )**

MCEnableAxis

MCEnableAxis( ) turns the specified axis on or off.

```
void MCEnableAxis(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   short int state                // Boolean flag for on/off setting of axis
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number to turn on or off.
state Flag to indicate if this axis should be turned on or turned off:

Value	Description
TRUE	Turn on *axis*.
FALSE	Turn off *axis*.

Returns
This function does not return a value.

Comments
This function does much more than just enable or disable *axis*. However, as the name implies, the selected axis(axes) will be turned on or off depending upon the value of *state*. Note that an axis must be enabled before any motion will take place. Issuing this command with *axis* set to MC_ALL_AXES will enable or disable all axes installed on *hCtlr*.

state will accept any non-zero value as TRUE, and will work correctly with most programming languages, including those that define TRUE as a non-zero value other than one (one is the Windows default value for TRUE).

If *axis* is off and then turned on, the following events will occur.

- The target and optimal positions are set to the present encoder position.
- The offset from **MCFindEdge( )**, **MCFindIndex( )** or **MCIndexArm( )** is applied.
- The data passed by **MCSetScale( )** are applied.
- MC_STAT_AMP_ENABLE will be set.
- MC_STAT_AMP_FAULT, if present, will be cleared.
- MC_STAT_ERROR, if present, will be cleared.
- MC_STAT_FOLLOWING, if present, will be cleared.
- MC_STAT_MLIM_TRIP, if present, will be cleared.
- MC_STAT_MSOFT_TRIP, if present, will be cleared.
- MC_STAT_PLIM_TRIP, if present, will be cleared.

- MC_STAT_PSOFT_TRIP, if present, will be cleared.

If *axis* is on and then turned on again, the following events will occur.

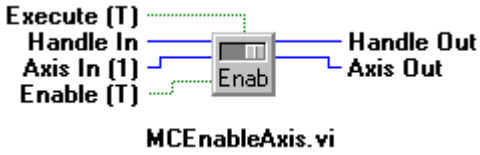
- The offset from **MCFindEdge( )**, **MCFindIndex( )** or **MCIndexArm( )** is applied.
- The data passed by **MCSetScale( )** are applied.

Calling this function to enable or disable an axis while it is in motion is not recommended. However, should it be done, *axis* will cease the current motion profile, and MC_STAT_AT_TARGET will be set.

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes
Delphi: procedure MCEnableAxis(hCtlr: HCTRLR; axis: Word; state: SmallInt); stdcall;
VB: Sub MCEnableAxis (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal state As Integer)
LabVIEW:



MCCL Reference
MF, MN

See Also
MCAbort( ), **MCStop( )**

MCEnableBacklash

MCEnableBacklash( ) sets the backlash compensation distance and turns backlash compensation on or off, depending upon the value of *state*.

```
long int MCEnableBacklash(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double backlash,               // backlash compensation distance
   short int state                // enable state
);
```

Parameters

hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number to control the backlash setting of.
backlash Amount of backlash compensation to apply. This parameter is ignored, if *state* is FALSE.
state Specifies whether the channel is to be turned on or turned off.

Value	Description
TRUE	Turns backlash compensation on.
FALSE	Turns backlash compensation off.

Returns

This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments

In applications where the mechanical system is not directly connected to the motor, it may be required that the motor move an extra amount to take up gear backlash. The *backlash* parameter to this function sets the amount of this compensation, and should be equal to one half of the amount the axis must move to take up the backlash when it changes direction.

state will accept any non-zero value as TRUE, and will work correctly with most programming languages, including those that define TRUE as a non-zero value other than one (one is the Windows default value for TRUE).

Compatibility

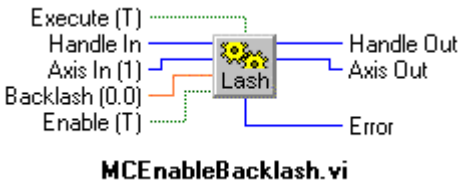
Stepper axes, the DC2, DCX-PC, and DCX-PCI100 controllers do not support backlash compensation.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 2.0 or higher

Prototypes

Delphi: function MCEnableBacklash(hCtlr: HCTRLR; axis: Word; backlash: Double; state: SmallInt): Longint; stdcall;
VB: Function MCEnableBacklash(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal backlash As Double, ByVal state As Integer) As Long
LabVIEW:



MCCL Reference

BD, BF, BN

MCEnableCapture

MCEnableCapture( ) begins position capture for the specified axis if *count* is greater than zero, or stops position capture if *count* is zero.

```
long int MCEnableCapture (
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   long int count                 // number of points to capture
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to begin or end position capture.
count	Set to zero to disable capture mode, or to a number greater than zero to capture that many positions.

Returns

This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments

This functions enables the high-speed capture of count points (maximum 512) if count is greater than zero, or disables position capture if *count* is -1. The count of currently captured data points may be obtained using **MCGetCount( )**, and captured position values may be retrieved using **MCGetCaptureData( )**.

Compatibility

The DC2 stepper axes, and the MC100, MC110, MC150, MC160 modules when installed on the DCX-PC100 controller do not support data capture. The DCX-PCI100 controller does not support torque mode nor do any stepper axes, which prevents the capture of torque values.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 3.1 or higher

Prototypes

Delphi:	function MCEnableCapture(hCtlr: HCTRLR; axis: Word; count: Longint): Longint; stdcall;
VB:	Function MCEnableCapture(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal count As Long) As Long
LabVIEW:	Not Supported

MCCL Reference

CB

See Also

MCGetCaptureData( ), **MCGetCount( )**

MCEnableCompare

MCEnableCompare( ) enables or disables high-speed compare mode for the specified axis.

```
long int MCEnableCompare(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   long int flag                  // flag to enable/disable compare state
);
```

Parameters

hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number to enable high-speed compare.
flag Flag to indicate if this axis should be turned on or turned off:

Value	Description
MC_COMPARE_DISABLE	Disable high-speed compare for Axis.
MC_COMPARE_ENABLE	Enable high-speed compare for Axis.

Returns

This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments

The high-speed compare function for *axis* is enabled or disabled by this function. High-speed compare mode must first be initialized by **MCConfigureCompare( )** before compare mode may be enabled. To determine how many compares have occurred use **MCGetCount( )**.

Compatibility

The DC2, DCX-PC100, DCX-AT200, and DCX-PCI100 controllers do not support high-speed position compare.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 3.1 or higher

Prototypes

Delphi: function MCEnableCompare(hCtlr: HCTRLR; axis: Word; flag: Longint): Longint; stdcall;
VB: Function MCEnableCompare(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal flag As Long) As Long
LabVIEW: Not Supported

MCCL Reference

BC

See Also

MCConfigureCompare( ), **MCGetCount( )**

MCEnableDigitalFilter

MCEnableDigitalFilter( ) enables or disables the digital filter capability of advanced motor modules, such as the MC300.

```
long int MCEnableDigitalFilter(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   long int state                 // Boolean flag enables/disables digital
                                  // filter
);
```

Parameters

hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number to enable digital filter.
state Flag to indicate if digital filter should be enabled on or disabled:

Value	Description
TRUE	Enable digital filter for *axis*.
FALSE	Disable digital filter for *axis*.

Returns

This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments

The digital filter function for *axis* is enabled or disabled by this function. Digital filter coefficients are loaded using **MCSetDigitalFilter( )** and may be read back from the controller using **MCGetDigitalFilter( )**. The function **MCIsDigitalFilter( )** will return a flag indicating the current enabled state of the digital filter, and **MCGetCount( )** may be used to determine the maximum filter size and the size of the currently loaded filter.

state will accept any non-zero value as TRUE, and will work correctly with most programming languages, including those that define TRUE as a non-zero value other than one (one is the Windows default value for TRUE).

Compatibility

The DC2, DCX-PC100, DCX-AT200, DCX-PCI100 controllers, MC360 and MC362 modules do not support digital filtering.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 3.1 or higher

Prototypes

Delphi: function MCEnableDigitalFilter(hCtlr: HCTRLR; axis: Word; state: Longint): Longint; stdcall;
VB: Function MCEnableDigitalFilter(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal state As Long) As Long
LabVIEW: Not Supported

MCCL Reference

NF, YF

See Also
MCGetCount( ), **MCGetDigitalFilter( )**, **MCIsDigitalFilter( )**, **MCSetDigitalFilter( )**

MCEnableGearing

MCEnableGearing( ) enables or disables electronic gearing for the specified *axis* / *master* pair.

```
void MCEnableGearing(
   HCTRLR hCtlr,                // controller handle
   WORD axis,                   // axis number
   WORD master,                 // master axis number
   double ratio,                // gearing ratio
   short int state              // enable state
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number for which to enable or disable gearing.
master Master axis that *axis* is to follow.
ratio Ratio at which *axis* is to reproduce *master*'s motions.
state Specifies whether the gearing is to be enabled on or disabled.

Value	Description
TRUE	Enables gearing.
FALSE	Disables gearing.

Returns
This function does not return a value.

Comments
This function permits you to configure one axis to automatically reproduce the motions of a master axis. In addition, by using a ratio of other than 1.0, the reproduced motion can be scaled as desired.

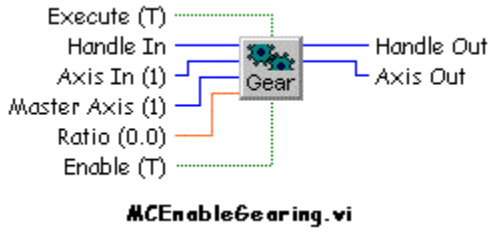
DC2 users should express the ratio as a floating point value (i.e. 0.5 for 2:1, 2.0 for 1:2, etc.). **MCEnableGearing( )** automatically converts this ratio to the 32 bit fixed point fraction the DC2 requires. The DCX-PC100 controller supports only a fixed ration of 1:1, the Ratio parameter is ignored for this controller.

state will accept any non-zero value as TRUE, and will work correctly with most programming languages, including those that define TRUE as a non-zero value other than one (one is the Windows default value for TRUE).

Compatibility
The DCX-PCI100 controller, DC2 stepper axes, the MC150, MC160, MC200, and MC260 modules when placed on the DCX-PC100 controller do not support gearing.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes
Delphi: procedure MCEnableGearing(hCtlr: HCTRLR; axis, master: Word; ratio: Double; state: SmallInt); stdcall;
VB: Sub MCEnableGearing(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal master As Integer, ByVal ratio As Double, ByVal state As Integer)
LabVIEW:



MCCL Reference
SM, SS

MCEnableJog

MCEnableJog( ) function enables or disables jogging for the axis specified by *axis*.

```
void MCEnableJog(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   short int state                // enable state
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number for which to enable or disable synchronized motion.
state Specifies whether the synchronized motion is to be enabled on or disabled.

Value	Description
TRUE	Enables synchronized motion.
FALSE	Disables synchronized motion.

Returns
This function does not return a value.

Comments
The selected *axis* should be configured for jogging using the **MCSetJogConfig( )** function before being enabled by this function.

state will accept any non-zero value as TRUE, and will work correctly with most programming languages, including those that define TRUE as a non-zero value other than one (one is the Windows default value for TRUE).

Compatibility
The DCX-PCI controllers, DC2 stepper axes, MC150, and MC160 modules do not support jogging.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes
Delphi: procedure MCEnableJog(hCtlr: HCTRLR; axis: Word; state: SmallInt); stdcall;
VB: Sub MCEnableJog(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal state As Integer)
LabVIEW: Not Supported

MCCL Reference
JF, JN

See Also
MCGetJogConfig( ), **MCSetJogConfig( )**

MCEnableSync

MCEnableSync( ) enables or disables synchronized motion for contour path motion for the specified axis.

```
void MCEnableSync(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   short int state                // enable state
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number for which to enable or disable synchronized motion.
state Specifies whether the synchronized motion is to be enabled on or disabled.

Value	Description
TRUE	Enables synchronized motion.
FALSE	Disables synchronized motion.

Returns
This function does not return a value.

Comments

This function is issued to the controlling axis of a contour path motion, prior to issuing any contour path motions, to inhibit any motion until a call to **MCGoEx( )** is made.

state will accept any non-zero value as TRUE, and will work correctly with most programming languages, including those that define TRUE as a non-zero value other than one (one is the Windows default value for TRUE).

Compatibility

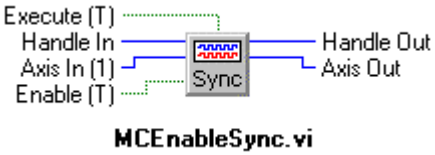
The MCAPI does not does not support contouring on the DC2, DCX-PC100, or DCX-PCI100 controllers.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure MCEnableSync(hCtlr: HCTRLR; axis: Word; state: SmallInt); stdcall;
VB: Sub MCEnableSync(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal state As Integer)
LabVIEW:



MCCL Reference

NS, SN

See Also

MCGoEx( )

MCFindAuxEncIdx

MCFindAuxEncIdx( ) arms the auxiliary encoder index capture function of an axis.

```
long int MCFindAuxEncIdx(
   HCTRLR hCtlr,                // controller handle
   WORD axis,                   // axis number
   double position              // reserved for future use
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number for which to search for the index signal.
position	This parameter is ignored by current motion controller firmware.

Returns
This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments
This function arms the auxiliary encoder index capture function of an axis. The function remains pending until the auxiliary encoder index input of the module goes active, at which point, MC_STAT_INP_AUX will be latched. This function does not cause any motion to be started or stopped.

A homing routine may incorporate this function by using **MCDecodeStatus**( ) to determine when MC_STAT_INP_AUX latches. After making sure the axis has stopped, you may determine how far the current position is from where the auxiliary encoder index occurred. The difference between **MCGetAuxEncPosEx**( ) and **MCGetAuxEncIdxEx( )** should be used as the current position through a call to **MCSetAuxEncPos( )**.

At this time, the firmware does not support the *position* parameter. We advise you set *position* to zero, so that future firmware updates will not break your code.

Compatibility
The DC2, DCX-PCI100 controllers, MC100, MC110, MC150, and MC320 modules do not support auxiliary encoders. Closed-loop steppers do not support auxiliary encoder functions, since the connected encoder is considered a primary encoder.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 2.2 or higher

Prototypes
Delphi: function MCFindAuxEncIdx(hCtlr: HCTRLR; axis: Word; position: Double): Longint; stdcall;
VB: Function MCFindAuxEncIdx(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal position As Double) As Long
LabVIEW: Not Supported

MCCL Reference
AF

See Also
MCBlockBegin( ), **MCFindIndex( )**, **MCGetAuxEncIdxEx( )**

MCFindEdge

MCFindEdge( ) is used to initialize a motor at a given position, relative to the home or coarse home input.

```
long int MCFindEdge (
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double position                // new position for edge
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number for which to search for the edge signal.
position	The position where the edge signal is sensed for the axis will be set to *position* after a call to **MCEnableAxis( )**.

Returns

This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments

This function is used to initialize a motor at a given position. The function remains pending until the home input of the module goes active. This function does not cause any motion to be started or stopped. See the example code in the online help for details of how to use **MCFindEdge( )**.

Once this command is issued, the calling program will not be able to communicate with the board until the home input is seen as high for *axis*. We recommend using **MCEdgeArm( )** and **MCIsEdgeFound( )** instead.

Only after an **MCEnableAxis( )** call will the position where the home input was seen as high for *axis* be set to the value of the *position* parameter.

The DC2 controllers, MC100, MC110, and MC260 modules use coarse home instead of home, but this still translates to MC_STAT_INP_HOME. In these cases, **MCDecodeStatus( )** should be used instead of this function.

Compatibility

The DC2 stepper axes, MC200 and MC210 when installed on the DCX-AT200, MC300, MC302, and MC320 modules do not support this command.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 2.0 or higher

Prototypes

Delphi:	function MCFindEdge(hCtlr: HCTRLR; axis: Word; position: Double): Longint; stdcall;
VB:	Function MCFindEdge Lib(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal position As Double) As Long
LabVIEW:	Not Supported

MCCL Reference

FE

See Also
MCBlockBegin( ), **MCEdgeArm( )**, **MCFindIndex( )**, **MCIsEdgeFound( )**, **MCWaitForEdge( )**

MCFindIndex

MCFindIndex( ) is used to initialize a servo or closed-loop stepper motor at a given position, relative to the index input.

```
long int MCFindIndex(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double position                // new position for index
);
```

Parameters
hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number for which to search for the index signal.
position	The position where the encoder index pulse occurred for the axis will be set to *position* after a call to **MCEnableAxis( )**.

Returns
This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments
This function is used to initialize a servo motor at a given position. The function remains pending until the index input of the module goes active. This function does not cause any motion to be started or stopped. See the example code in the online help for details of how to use **MCFindIndex( )**.

Once this command is issued, the calling program will not be able to communicate with the board until the *axis* captures the encoder index. We recommend instead using and confirming that **MCIndexArm( )** has captured the index through **MCIsIndexFound( )** before calling **MCWaitForIndex( )** to avoid this problem.

Only after an **MCEnableAxis( )** call will the position where the encoder index pulse occurred for *axis* be set to the value of the *position* parameter.

Compatibility
Open-loop stepper axes do not support this command, since the connected encoder is considered an auxiliary encoder.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 2.0 or higher

Prototypes

Delphi: function MCFindIndex(hCtlr: HCTRLR; axis: Word; position: Double): Longint; stdcall;
VB: Function MCFindIndex(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal position As Double) As Long
LabVIEW: Not Supported

MCCL Reference

FI

See Also

MCBlockBegin( ), **MCFindAuxEncIdx( )**, **MCFindEdge( )**, **MCIndexArm( )**, **MCWaitForEdge( )**, **MCWaitForIndex( )**

MCGoEx

MCGoEx( ) initiates a motion when operating in velocity mode.

```
long int MCGoEx(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double param                   // optional argument for the GO command
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to command.
param	Argument to the GO command.

Returns

This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments

The axis must be configured for velocity mode operation before issuing a **MCGoEx( )** call. All axes may be instructed to move by setting the Axis parameter to MC_ALL_AXES.

To enable cubic splining while in contour mode on the DCX-AT200 or DCX-AT300 use **MCGoEx( )** with the value of *param* set to 1.0.

Compatibility

There are no compatibility issues with this function.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 2.1 or higher

Prototypes

Delphi: function MCGoEx(hCtlr: HCTRLR; axis: Word; param: Double): Longint; stdcall;
VB: Function MCGoEx(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal param As Double) As Long
LabVIEW:

MCCL Reference

GO

See Also

MCSetOperatingMode( ), **MCStop( )**

MCGoHome

MCGoHome( ) initiates a home motion for the specified axis or all axes.

```
void MCGoHome(
   HCTRLR hCtlr,                  // controller handle
   WORD axis                      // axis number
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to command.

Returns

This function does not return a value.

Comments

The home or zero position is used that was last set by calling **MCSetPosition( )**. This command effectively executes a **MCMoveAbsolute( )** with a target position of 0.0.

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility

There are no compatibility issues with this function.

Requirements

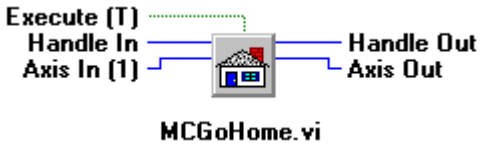
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure MCGoHome(hCtlr: HCTRLR; axis: Word); stdcall;

VB: Sub MCGoHome Lib(ByVal hCtrlr As Integer, ByVal axis As Integer)

LabVIEW:



MCCL Reference

GH

See Also

MCMoveAbsolute( ), **MCSetPosition( )**

MCIndexArm

MCIndexArm( ) arms the index capture function of a servo or closed-loop stepper axis.

```
long int MCIndexArm(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double position                // new position for index
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number for which to search for the index signal.
position	The position where the encoder index pulse occurred for the axis will be set to *position* after a call to **MCEnableAxis( )**.

Returns

This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments

This function is used to initialize a servo motor to a specified position where the encoder index pulse occurs. The function remains pending until the encoder index input of the module goes active, after which a call to **MCEnableAxis( )** sets the position where the encoder index pulse occurred to the value of the *position* parameter. This function does not cause any motion to be started or stopped.

For stepper axes this function performs in a similar fashion. The difference is that the stepper axis uses the home input signal in place of the encoder index input signal.

Only after an **MCEnableAxis( )** call will the position where the encoder index pulse occurred for *axis* be set to the value of the *position* parameter.

Compatibility

Open-loop stepper axes do not support this command, since the connected encoder is considered an auxiliary encoder.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 2.2 or higher

Prototypes

Delphi: function MCIndexArm(hCtlr: HCTRLR; axis: Word; position: Double): Longint; stdcall;
VB: Function MCIndexArm(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal position As Double) As Long
LabVIEW: Not Supported

MCCL Reference

IA

See Also

MCBlockBegin( ), **MCFindAuxEncIdx( )**, **MCFindIndex( )**, **MCWaitForIndex( )**

MCLearnPoint

MCLearnPoint( ) stores the current actual position or target position for the specified *axis* in point memory at location specified by *index*.

```
long int MCLearnPoint(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   WORD index,                    // point memory index
   WORD mode                      // type of position to store
);
```

Parameters

hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number to store data for.
index Storage location for point data.
mode Determines if the actual position or the target position will be stored:

Value	Description
MC_LRN_POSITION	Learns the current actual position for the specified axis.
MC_LRN_TARGET	Learns the current target position for the specified axis.

Returns
This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments
The actual position of an axis may be stored as it is moved; or, by disabling the axis, position commands may be issued to the axis, and the target positions stored, without actually moving the axis (see online help examples).

The number of points that may be stored will vary with the number of motor axes installed and the type of controller (see the compatibility section, below, for controller dependent limits). The first storage is location zero (not location 1).

The current position of all axes may be stored by setting the Axis parameter to MC_ALL_AXES.

Compatibility
The number of points that can be stored is dependent on the controller type and in some cases on the number of installed axes:

Controller	1	2	3	4	5	6	7	8
DCX-PCI300	256	256	256	256	256	256	256	256
DCX-PCI100	256	256	256	256	256	256	256	256
DCX-AT300	1536	768	512	384	307	256	n/a	n/a
DCX-AT200	1536	768	512	384	307	256	n/a	n/a
DCX-PC100	4096	2048	1365	1024	819	682	585	512
DC2-PC100	n/a	2048	n/a	n/a	n/a	n/a	n/a	n/a
DCX-PCI300	256	256	256	256	256	256	256	256
DCX-PCI100	256	256	256	256	256	256	256	256
DCX-AT300	1536	768	512	384	307	256	n/a	n/a
DCX-AT200	1536	768	512	384	307	256	n/a	n/a
DCX-PC100	4096	2048	1365	1024	819	682	585	512

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes
Delphi: function MCLearnPoint(hCtlr: HCTRLR; axis: Word; index: Longint; mode: Word): Longint; stdcall;
VB: Function MCLearnPoint Lib(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal index As Long, ByVal mode As Integer) As Long
LabVIEW: Not Supported

MCCL Reference
LP, LT

See Also
MCMoveToPoint( )

MCMoveAbsolute

MCMoveAbsolute( ) initiates an absolute position move for the specified axis.

```
void MCMoveAbsolute(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double position                // new absolute position
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to move.
position	Absolute position to move to.

Returns

This function does not return a value.

Comments

The axis must be enabled prior to executing a move (an exception to this is when the **MCMoveAbsolute( )** is used with **MCLearnPoint( )** in target mode).

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility

There are no compatibility issues with this function.

Requirements

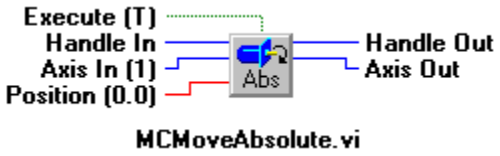
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure MCMoveAbsolute(hCtlr: HCTRLR; axis: Word; position: Double); stdcall;

VB: Sub MCMoveAbsolute(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal position As Double)

LabVIEW:



MCCL Reference

MA

See Also

MCMoveRelative( ), **MCSetPosition( )**

MCMoveRelative

MCMoveRelative( ) initiates a relative position move for the specified axis or all axes.

```
void MCMoveRelative(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double distance                // distance to move from current position
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to move.
distance	Amount of distance to move.

Returns

This function does not return a value.

Comments

The axis must be enabled prior to executing a move (an exception to this is when the **MCMoveRelative( )** is used with **MCLearnPoint( )** in target mode).

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility

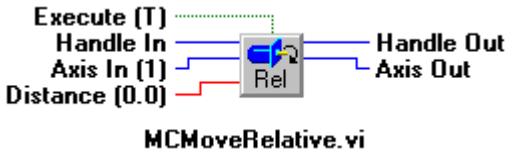
There are no compatibility issues with this function.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure MCMoveRelative(hCtlr: HCTRLR; axis: Word; distance: Double); stdcall;
VB: Sub MCMoveRelative(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal distance As Double)
LabVIEW:



MCCL Reference

MR

See Also

MCMoveAbsolute( ), **MCSetPosition( )**

MCMoveToPoint

MCMoveToPoint( ) initiates an absolute move to a stored location for the specified axis or all axes.

```
long int MCMoveToPoint(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   WORD index                     // index of point to move to
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to move.
index	Index of stored location to move to.

Returns

This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments

The motor must be enabled prior to executing a **MCMoveToPoint( )** and the point specified by *index* must have been stored by a previous call to **MCLearnPoint( )**. All axes may be instructed to move by setting the *axis* parameter to MC_ALL_AXES.

Compatibility

The DC2 stepper axes do not support this command.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi:	function MCMoveToPoint(hCtlr: HCTRLR; axis: Word; index: Longint): Longint; stdcall;
VB:	Function MCMoveToPoint Lib(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal index As Long) As Long
LabVIEW:	Not Supported

MCCL Reference

MP

See Also

MCLearnPoint( )

MCReset

MCReset( ) performs a complete reset of the axis or controller, leaving the specified axis (or axes) in the disabled state.

```
void MCReset(
   HCTRLR hCtlr,                  // controller handle
   WORD axis                      // axis number
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to reset.

Returns

This function does not return a value.

Comments

Setting the *axis* parameter to MC_ALL_AXES will cause the specified controller to be reset.

If you have enabled the hardware reset feature of the DCX-AT, or DCX-PC100 controllers **MCReset( )** will perform a hard reset when *axis* is equal to MC_ALL_AXES, or a soft reset when Axis specifies a particular axis. If this feature is off (the default state), **MCReset( )** issues the "RT" command to the board to perform any reset (this is a "soft" reset). On the DCX-AT200 and DCX-AT300 you must set jumper JP2 to connect pins 1 and 2 if Hard Reset is enabled, or connect pins 5 and 6 (factory default) if Hard Reset is disabled. On the DCX-PC100 you must set jumper JP4 to connect pins 1 and 2 if Hard Reset is enabled, or connect pins 5 and 6 (factory default) if Hard Reset is disabled. See the Motion Control Panel online help for how to enable the MCAPI Hardware Reset feature.

Compatibility

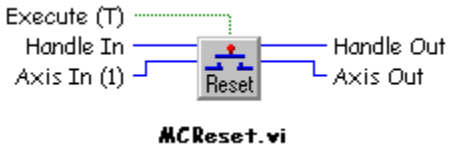
The DC2 series, DCX-PC100, DCX-AT100, and DCX-AT200 (prior to firmware version 1.2a) controllers do not support the resetting of individual axes. In these cases when this command is executed, the *axis* parameter is ignored and a controller reset is performed.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure MCReset(hCtlr: HCTRLR; axis: Word); stdcall;
VB: Sub MCReset Lib(ByVal hCtrlr As Integer, ByVal axis As Integer)
LabVIEW:



MCCL Reference

RT

See Also

MCAbort( ), **MCStop( )**

MCStop

MCStop( ) stops the specified axis or axes using the pre-programmed deceleration values.

```
void MCStop(
   HCTRLR hCtlr,                  // controller handle
   WORD axis                      // axis number
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to stop.

Returns

This function does not return a value.

Comments

This function initiates a controlled axis stop, as compared with **MCAbort( )** which stops the axis abruptly.

Following a call to **MCStop( )** verify that the axis has stopped using or **MCIsStopped( )** or **MCWaitForStop( )**. Then call **MCEnableAxis( )** prior to issuing another motion command.

Following a call to **MCStop( )** on the DCX-PC100 controller when in velocity mode, call **MCSetOperatingMode( )** prior to issuing another motion command.

Compatibility

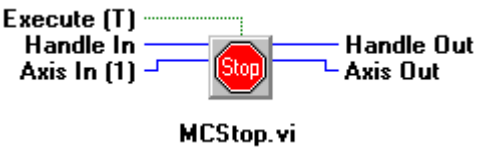
There are no compatibility issues with this function.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure MCStop(hCtlr: HCTRLR; axis: Word); stdcall;
VB: Sub MCStop(ByVal hCtrlr As Integer, ByVal axis As Integer)
LabVIEW:



MCCL Reference

ST

See Also

MCAbort( ), MCEnableAxis( ), MCIsStopped( ), MCSetOperatingMode( ), MCWaitForStop( )

MCWait

MCWait( ) waits the specified number of seconds before returning to the caller.

```
void MCWait(
   HCTRLR hCtlr,                 // controller handle
   double period                 // length of delay
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
period	Length of delay, in seconds.

Returns

This function does not return a value.

Comments

The delay is specified in seconds, unless **MCSetScale( )** has been called to change the time scale.

Once this command is issued, the calling program will not be able to communicate with the board until *period* elapses. We recommend creating your own time based looping structure.

Compatibility

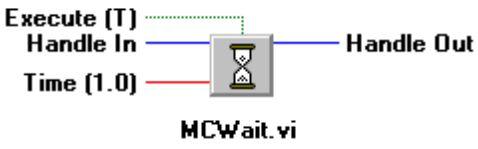
There are no compatibility issues with this function.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure MCWait(hCtlr: HCTRLR; period: Double); stdcall;
VB: Sub MCWait(ByVal hCtrlr As Integer, ByVal period As Double)
LabVIEW:



MCCL Reference

WA

See Also

MCWaitForPosition( ), **MCWaitForRelative( )**, **MCWaitForStop( )**, **MCWaitForTarget( )**

MCWaitForEdge

MCWaitForEdge( ) waits for the coarse home input to go to the specified logic level for a servo, closed-loop stepper, or an MC260 open-loop stepper. When used with an open-loop stepper (excluding an MC260) this function completes a call to **MCEdgeArm( )**. Note that when used with an open-loop stepper (excluding an MC260), the parameter *state* has no effect.

```
long int MCWaitForEdge(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   short int state                // selects logic level to wait for
);
```

Parameters

hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number to wait for.
state Selects the coarse home logic level to wait for:

Value	Description
TRUE	Wait for coarse home to go active.
FALSE	Wait for coarse home to go inactive.

Returns

This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments

This function behaves differently depending on what type of module *axis* is and whether it is in open-loop or closed-loop mode. In both cases instruction processing is paused until the home or coarse home input, respectively, goes to the specified logic state. In open-loop mode, this function is one of three functions that must be called to set the home input signal transition to a predetermined position. In closed-loop mode, this function is used to find a home sensor to qualify an index pulse on servo or closed-loop stepper. However, using this function with a closed-loop system is discouraged.

In open-loop mode, exclusively stepper modules (excluding the MC260, see the closed-loop section for function behavior), this function should be called after **MCIsEdgeFound( )** confirms that the home input has latched from a previous call to **MCEdgeArm( )**. After this function returns control to the calling program, a call to **MCEnableAxis( )** will apply *position* defined in **MCEdgeArm( )** to the position where the home input first latched.

Once this command is issued, the calling program will not be able to communicate with the board until the home input signal is detected. We recommend calling **MCIsEdgeFound( )**, to confirm the home input is active prior to calling this function.

Note that when used with an open-loop stepper (excluding an MC260), the parameter *state* has no effect. Also, this function is only looking for an active signal state, not a transition.

When a module used in closed-loop mode or with an MC260, this function is called by itself to return when the home input state level defined by *state* is observed. To assure a leading or trailing edge, this function would have to be called twice with *state* different in both cases.

Once this command is issued, the calling program will not be able to communicate with the board until *state* matches the coarse home logic level. We recommend creating your own looping structure based on **MCDecodeStatus( )** and MC_STAT_INP_HOME instead of using this function.

state will accept any non-zero value as TRUE, and will work correctly with most programming languages, including those that define TRUE as a non-zero value other than one (one is the Windows default value for TRUE).

See the example code in the online help for details of how to use **MCWaitForEdge( )**.

Compatibility
The DC2 stepper axes, MC150, and MC160 modules do not support this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 2.0 or higher

Prototypes
Delphi: function MCWaitForEdge(hCtlr: HCTRLR; axis: Word; state: SmallInt): Longint; stdcall;
VB: Function MCWaitForEdge(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal state As Integer) As Long
LabVIEW: Not Supported

MCCL Reference
WE

See Also
MCEdgeArm( ), **MCFindEdge( )**, **MCFindIndex( )**, **MCIsEdgeFound( )**

MCWaitForIndex

MCWaitForIndex( ) waits until the index pulse has been observed on servo or closed-loop stepper axis.

```
long int MCWaitForIndex(
   HCTRLR hCtlr,                  // controller handle
   WORD axis                      // axis number
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number to wait for.

Returns
This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments
This function is used to initialize a motor to a given position relative to the index pulse. When called after **MCIndexArm( )**, it provides the exact same functionality as **MCFindIndex( )**. The benefit is that you may query the controller through **MCIsIndexFound( )** to see that the index has latched. Once the index has been seen, a call to **MCWaitForIndex( )** will not cause the board to stop communicating where **MCFindIndex( )** has the potential to cause the controller to stop communicating.

Once this command is issued, the calling program will not be able to communicate with the board until *axis* captures the encoder index. We recommend confirming that **MCIndexArm( )** has captured the index by using **MCIsIndexFound( )** before calling **MCWaitForIndex( )** to avoid this problem.

Only after an **MCEnableAxis( )** call will the position where the encoder index pulse occurred for *axis* be set to the value of the *position* parameter.

Compatibility
Open-loop stepper axes do not support this command, since the connected encoder is considered an auxiliary encoder.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 2.2 or higher

Prototypes
Delphi: function MCWaitForIndex(hCtlr: HCTRLR; axis: Word): Longint; stdcall;
VB: Function MCWaitForIndex(ByVal hCtrlr As Integer, ByVal axis As Integer) As Long
LabVIEW: Not Supported

MCCL Reference
WI

See Also
MCFindAuxEncIdx( ), **MCFindEdge( )**, **MCFindIndex( )**, **MCIndexArm( )**, **MCIsIndexFound( )**

MCWaitForPosition

MCWaitForPosition( ) waits for the *axis* to reach the specified *position* before allowing the next command to execute.

```
void MCWaitForPosition(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double position                // position to wait for
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to wait on to reach specified position.
position	Absolute position to wait for.

Returns

This function does not return a value.

Comments

You must start the specified *axis* moving, and make certain the motion will at least reach the wait position, in order for this function to return to the calling program.

Once this command is issued, the calling program will not be able to communicate with the board until *axis*' encoder reaches *position*.

Compatibility

The DC2 stepper axes, MC150, and MC160 modules do not support this function.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi:	procedure MCWaitForPosition(hCtlr: HCTRLR; axis: Word; position: Double); stdcall;
VB:	Sub MCWaitForPosition(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal position As Double)
LabVIEW:	Not Supported

MCCL Reference

WP

See Also

MCWait( ), **MCWaitForRelative( )**, **MCWaitForStop( )**, **MCWaitForTarget( )**

MCWaitForRelative

MCWaitForRelative( ) waits for the *axis* to reach a position that is specified relative to the target position.

```
void MCWaitForRelative(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double distance                // relative position to wait for
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to wait on for to reach specified position.
distance	Position, relative to the current target position, to wait for.

Returns

This function does not return a value.

Comments

You must start the specified *axis* moving, and make certain the motion will at least reach the wait position, in order for this function to return to the calling program. The position argument is specified as a distance from the target position.

Once this command is issued, the calling program will not be able to communicate with the board until *axis*' encoder traverses *distance*.

Compatibility

The DC2 stepper axes, MC150, and MC160 modules do not support this function.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure MCWaitForRelative(hCtlr: HCTRLR; axis: Word; distance: Double); stdcall;
VB: Sub MCWaitForRelative(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal distance As Double)
LabVIEW: Not Supported

MCCL Reference

WR

See Also

MCWait( ), **MCWaitForPosition( )**, **MCWaitForStop( )**, **MCWaitForTarget( )**

MCWaitForStop

MCWaitForStop( ) waits for the specified *axis* or all axes to come to a stop. An optional dwell after the stop may be specified within this command to allow the mechanical system to come to rest.

```
void MCWaitForStop(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double dwell                   // dwell time after stop
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number function is waiting for to stop.
dwell	Delay time after stop has occurred.

Returns

This function does not return a value.

Comments

MCWaitForStop( ) is necessary for synchronizing motions, and for making certain that a prior motion has completed before beginning a new motion.

Once this command is issued, the calling program will not be able to communicate with the board until *axis*' encoder comes to rest. We recommend using **MCIsStopped( )** or **MCIsAtTarget( )** instead.

Compatibility

There are no compatibility issues with this function.

Requirements

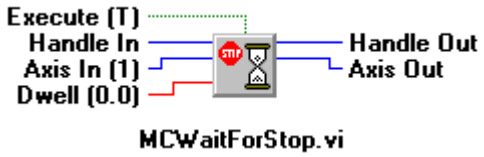
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure MCWaitForStop(hCtlr: HCTRLR; axis: Word; dwell: Double); stdcall;

VB: Sub MCWaitForStop(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal dwell As Double)

LabVIEW:



MCCL Reference

WS

See Also

MCIsAtTarget( ), **MCIsStopped( )**, **MCWait( )**, **MCWaitForPosition( )**, **MCWaitForRelative( )**, **MCWaitForTarget( )**

MCWaitForTarget

MCWaitForTarget( ) waits for the specified *axis* to reach its target position. An optional dwell after the stop may be specified within this command to allow the mechanical system to come to rest.

```
void MCWaitForTarget(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double dwell                   // dwell time after stop
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number function is waiting for to reach the target position.
dwell	Delay time after stop has occurred.

Returns

This function does not return a value.

Comments

For a servo axis to be considered "at target" it must remain within the **Deadband** region for the **DeadbandDelay** period. **Deadband** and **DeadbandDelay** are specified in the **MCMOTIONEX** configuration structure.

Once this command is issued, the calling program will not be able to communicate with the board until *axis*' encoder settles within the **Deadband** region for the **DeadbandDelay** period. We recommend using **MCDecodeStatus( )** along with MC_STAT_AT_TARGET instead.

Compatibility

The DC2 and DCX-PC100 controllers do not support this function.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure MCWaitForTarget(hCtlr: HCTRLR; axis: Word; dwell: Double); stdcall;
VB: Sub MCWaitForTarget(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal dwell As Double)
LabVIEW: Not Supported

MCCL Reference

WT

See Also

MCGetMotionConfigEx( ), **MCSetMotionConfigEx( )**, **MCWaitForPosition( )**, **MCWaitForRelative( )**, **MCWaitForStop( )**

Chapter Contents

Chapter 14

MCAPI Reporting Functions

Reporting functions allow the calling program to query the board to determine how parameters have been configured, as well as getting information regarding the position and status of any given axis.
Also included in this category are functions that allow the program to trap and decode errors.

To see examples of how the functions in this chapter are used, please refer to the online Motion Control API Reference.

MCDecodeStatus

MCDecodeStatus( ) permits you to test flags in the controller status word in a way that is independent of the type of controller being inspected.

```
long int MCDecodeStatus(
   HCTRLR hCtlr,                  // controller handle
   DWORD status,                  // status word
   long int bit                   // status bit selection flag
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
status	Status value returned from a previous call to **MCGetStatus( )**.
bit	Status bit to decode. Over fifty different status bit flags (not all flags are supported by all controllers) are defined in the Constants section of this help file. Valid Bit constants begin with "MC_STAT_".

Returns

This function returns TRUE if the selected bit is set. Otherwise, FALSE is returned if the bit is not set or the bit does not apply to this controller type.

Comments

Using this function to test the status word returned by **MCGetStatus( )** isolates the program from controller dependent bit ordering of the status word. The sample programs include numerous examples of the **MCDecodeStatus( )** function.

To assist with proper constant selection two tables have been provided with the online help. The Status Word Lookup Table lists the constants in the same order as the status word bits they represent for each controller model, and has been included in Appendix C. A second table, The Status Word Cross Reference, lists the controller models supported by each constant, and will only be found in the online help.

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes
Delphi: function MCDecodeStatus(hCtlr: HCTRLR; status, bit: Longint): Longint; stdcall;
VB: Function MCDecodeStatus(ByVal hCtrlr As Integer, ByVal status As Long, ByVal bit As Long) As Long
LabVIEW:

MCCL Reference
None

See Also
MCGetStatus( ), online help sample programs

MCErrorNotify

MCErrorNotify( ) registers with the MCAPI a specific window procedure that is to receive message based notification of API errors for this controller handle.

```
void MCErrorNotify(
   HWND hWnd,                    // error handling window procedure
   HCTRLR hCtlr,                 // controller handle
   DWORD errorMask               // mask to select error category
);
```

Parameters
hWnd Handle of window procedure to receive error messages.
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
errorMask Selects error categories to be notified about. Any combination of the MCERRMASK_xxxx constants may be OR'ed together to select errors to be

reported. The constant MCERRMASK_STANDARD includes the most common error messages.

Returns
This function does not return a value.

Comments
Only one window procedure at a time may receive error messages for a controller handle. If another window procedure attempts to hook the error messages for a handle that already has an error handler, it will replace the current error handler. In practice, this is not a problem as applications have control of the handle. They can decide who to have hook the error notification mechanism.

The error notification message is a pre-agreed upon, inter-application message that goes by the name "MCErrorNotify". Application programs need to call the Windows function **RegisterWindowMessage( )** with the message name "MCErrorNotify" to obtain the numeric value if the message. The error message will have a numeric error code as its *wParam*, and a pointer to a null-terminated ASCII string representation of the name of the function that caused the error as its *lParam*. The CWDemo sample application includes an example of hooking the error notification loop and processing error messages.

In the event of a bad controller handle passed to an API function as part of an API call, an error message will be broadcast to every windows procedure. This is done because with a bad handle there is no way for the API to identify which window procedure should receive the error. Rather than quietly tell no one, the API plays it safe and tells everyone.

The standard Windows message queue is small and may be over-run if error messages occur in rapid succession. During application development, when errors are most likely, you may want to call the Windows function **SetMessageQueue( )** in your **WinMain** function to set the application queue to something larger than the default size of 8 messages.

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.2 or higher

Prototypes
Delphi: procedure MCErrorNotify(hWnd: HWnd; hCtlr: HCTRLR; errorMask: Longint); stdcall;
VB: Sub MCErrorNotify(ByVal hWnd As Long, ByVal hCtrlr As Integer, ByVal errorMask As Long)
LabVIEW: Not Supported

MCCL Reference
None

See Also
MCGetError( ), **MCTranslateErrorEx( )**, CWDemo sample code

MCGetAccelerationEx

MCGetAccelerationEx( ) returns the current programmed acceleration value for the given axis, in whatever units the axis is configured for.

```
long int MCGetAccelerationEx(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double* pAccel                 // acceleration return value
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to query for acceleration
pAccel	Pointer to a double precision floating point variable that will hold the acceleration for the specified axis.

Returns

The acceleration value is placed in the variable specified by the pointer *pAccel* and MCERR_NOERROR is returned if there were no errors. If there was an error, one of the MCERR_xxxx error codes is returned and the variable pointed to by *pAccel* is left unchanged.

Comments

The acceleration value returned by this function is the same as the **Acceleration** field of the **MCMOTIONEX** structure returned by **MCGetMotionConfigEx( )**; **MCGetAccelerationEx( )** provides a short-hand method for obtaining just the acceleration value.

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility

The DC2 stepper axes do not support ramping.

Requirements

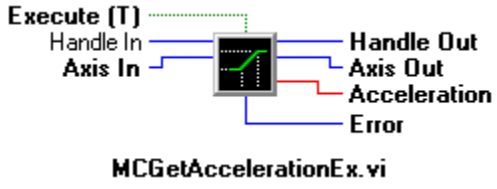
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes

Delphi: function MCGetAccelerationEx(hCtlr: HCTRLR; axis: Word; var pAccel: Double): Longint; stdcall;

VB: Function MCGetAccelerationEx(ByVal hCtrlr As Integer, ByVal axis As Integer, accel As Double) As Long

LabVIEW:



MCCL Reference

None

See Also
MCSetAcceleration( ), **MCGetMotionConfigEx( )**

MCGetAuxEncIdxEx

MCGetAuxEncIdxEx( ) returns the position where the auxiliary encoder's index pulse was observed.

```
long int MCGetAuxEncIdxEx(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double* pIndex                 // index position return value
);
```

Parameters
hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to query.
pIndex	Pointer to a double precision floating point variable that will hold the auxiliary encoder index position for the specified axis.

Returns
The auxiliary encoder index position is placed in the variable specified by the pointer *pIndex* and MCERR_NOERROR is returned if there were no errors. If there was an error, one of the MCERR_xxxx error codes is returned and the variable pointed to by *pIndex* is left unchanged.

Comments
The auxiliary encoder's position may be set (to zero) using the **MCSetAuxEncPos( )** function. The index position reported will be relative to this zero position.

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility
The DC2, DCX-PCI100 controllers, MC100, MC110, MC150, and MC320 modules do not support auxiliary encoders. Closed-loop steppers do not support auxiliary encoder functions, since the connected encoder is considered a primary encoder.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes
Delphi: function MCGetAuxEncIdxEx(hCtlr: HCTRLR; axis: Word; var pIndex: Double): Longint; stdcall;
VB: Function MCGetAuxEncIdxEx(ByVal hCtrlr As Integer, ByVal axis As Integer, index As Double) As Long
LabVIEW: Not Supported

MCCL Reference
AZ

See Also
MCFindAuxEncIdx( ), **MCGetAuxEncPosEx( )**, **MCSetAuxEncPos( )**

MCGetAuxEncPosEx

MCGetAuxEncPosEx( ) returns the current position of the auxiliary encoder.

```
long int MCGetAuxEncPosEx(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double* pPosition              // position return value
);
```

Parameters
hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to query.
pPosition	Pointer to a double precision floating point variable that will hold the auxiliary encoder position for the specified axis.

Returns
The auxiliary encoder position is placed in the variable specified by the pointer *pPosition* and MCERR_NOERROR is returned if there were no errors. If there was an error, one of the MCERR_xxxx error codes is returned and the variable pointed to by *pPosition* is left unchanged.

Comments
The auxiliary encoder's position may be set using the **MCSetAuxEncPos( )** function.

You may not set the *axis* parameter to MC_ALL_AXES for this command.

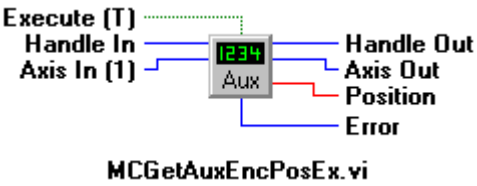
Compatibility
The DC2, DCX-PCI100 controllers, MC100, MC110, MC150, and MC320 modules do not support auxiliary encoders. Closed-loop steppers do not support auxiliary encoder functions, since the connected encoder is considered a primary encoder.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes
Delphi: function MCGetAuxEncPosEx(hCtlr: HCTRLR; axis: Word; var pPosition: Double): Longint; stdcall;

VB: Function MCGetAuxEncPosEx(ByVal hCtrlr As Integer, ByVal axis As Integer, position As Double) As Long

LabVIEW:



MCCL Reference
AT

See Also
MCGetAuxEncIdxEx( ), **MCSetAuxEncPos ( ))**

MCGetAxisConfiguration

MCGetAxisConfiguration( ) obtains the configuration for the specified axis. Configuration information includes the axis type, servo motor update rates, stepper motor step rates, etc.

```
long int MCGetAxisConfiguration(
   HCTRLR hCtlr,                 // controller handle
   WORD axis,                    // axis number
   MCAXISCONFIG* pAxisCfg        // address of axis configuration structure
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to query.
pAxisCfg	Points to an **MCAXISCONFIG** structure that receives the configuration information.

Returns
This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments
This function allows the application to query the driver about installed motor axis hardware and capabilities.

Before you call **MCGetAxisConfiguration( )** you must set the **cbSize** member to the size of the **MCAXISCONFIG** data structure. C/C++ programmers may use **sizeof( )**, Visual Basic and Delphi programmers will find current sizes for these data structures in the appropriate MCAPI.XXX header file.

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib

Version: MCAPI 3.0 or higher

Prototypes

Delphi: function MCGetAxisConfiguration(hCtlr: HCTRLR; axis: Word; var pAxisCfg: MCAXISCONFIG): Longint; stdcall;
VB: Function MCGetAxisConfiguration(ByVal hCtrlr As Integer, ByVal axis As Integer, axisCfg As MCAxisConfig) As Long
LabVIEW: Not Supported

MCCL Reference

Dual Port RAM

See Also

MCAXISCONFIG structure definition

MCGetBreakpointEx

MCGetBreakpointEx( ) returns the current breakpoint position as placed by the **MCWaitForPosition( )** or **MCWaitForRelative( )** command.

```
long int MCGetBreakpointEx(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double* pBreakpoint            // breakpoint position return value
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to query.
pBreakpoint	Pointer to a double precision floating point variable that will hold the breakpoint position for the specified axis.

Returns

The breakpoint position is placed in the variable specified by the pointer *pBreakpoint* and MCERR_NOERROR is returned if there were no errors. If there was an error, one of the MCERR_xxxx error codes is returned and the variable pointed to by *pBreakpoint* is left unchanged.

Comments

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility

The DCX-PC100 controller and stepper axes do not support this command.

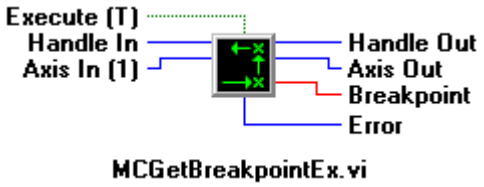
Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas

Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes

Delphi: function MCGetBreakpointEx(hCtlr: HCTRLR; axis: Word; var pBreakpoint: Double): Longint; stdcall;
VB: Function MCGetBreakpointEx(ByVal hCtrlr As Integer, ByVal axis As Integer, breakpoint As Double) As Long
LabVIEW:



MCCL Reference

TB

See Also

MCWaitForPosition( ), **MCWaitForRelative( )**

MCGetCaptureData

MCGetCaptureData( ) retrieves data collected following the most recent **MCCaptureData( )** call.

```
long int MCGetCaptureData(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number to get capture data from
   long int type,                 // type of capture data to retrieve
   long int start,                // index of starting point
   long int points,               // number of data points to retrieve
   double* pData                  // pointer to data array to for data
);
```

Parameters

hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number to query.
type Specifies the type of data to retrieve:

Value	Description
MC_CAPTURE_ACTUAL	Retrieves the captured actual position data.
MC_CAPTURE_ERROR	Retrieves the following error (difference between actual and optimal positions).
MC_CAPTURE_OPTIMAL	Retrieves the captured optimal position data.
MC_CAPTURE_TORQUE	Retrieves the captured torque data.

start Index of the first data point to retrieve. The index is zero based, i.e. the first data point is 0, not 1.

points	Total number of data points to retrieve.
pData	Pointer to a double precision floating point variable that will hold the breakpoint position for the specified axis.

Returns
This function places one or more captured data values in the array specified by the pointer *pData*, and MCERR_NOERROR is returned if there were no errors. If there was an error, one of the MCERR_xxxx error codes is returned and state of the array pointed to by *pData* is undefined.

Comments
Capture data settings (number of points, delay, etc.) are set with the **MCCaptureData( )** function.

Beginning with version 3.0 of the MCAPI users may use the **MCGetAxisConfiguration( )** function to determine the data capture capabilities of an axis.

Compatibility
The DC2 stepper axes, and the MC100, MC110, MC150, MC160 modules when installed on the DCX-PC100 controller do not support data capture. The DCX-PCI100 controller does not support torque mode nor do any stepper axes, which prevents the capture of torque values.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes
Delphi:	function MCGetCaptureData(hCtlr: HCTRLR; axis: Word; type, start, points: Longint; var pData: Double): Longint; stdcall;
VB:	Function MCGetCaptureData(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal start, ByVal argtype As Long, ByVal points As Long, data As Double) As Long
LabVIEW:	Not Supported

MCCL Reference
DO, DR, DQ

See Also
MCCaptureData( ), **MCGetAxisConfiguration( )**

MCGetContourConfig

MCGetContourConfig( ) obtains the contouring configuration for the specified axis.

```
long int MCGetContourConfig(
   HCTRLR hCtlr,                // controller handle
   WORD axis,                   // axis number
   MCCONTOUR* pContour          // structure to hold contour data
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to query.
pContour	Points to an **MCCONTOUR** structure that receives the configuration information for Axis.

Returns

The return value is TRUE if the function is successful. A return value of FALSE indicates the function did not find the Axis specified (*hCtlr* or *axis* incorrect).

Comments

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility

The MCAPI does not does not support contouring on the DC2, DCX-PC100, or DCX-PCI100 controllers.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi:	function MCGetContourConfig(hCtlr: HCTRLR; axis: Word; var pContour: MCCONTOUR): SmallInt; stdcall;
VB:	Function MCGetContourConfig Lib(ByVal hCtrlr As Integer, ByVal axis As Integer, contour As MCContour) As Integer
LabVIEW:	Not Supported

MCCL Reference

Controller RAM Motor Tables

See Also

MCSetContourConfig( ), **MCCONTOUR** structure definition

MCGetContouringCount

MCGetContouringCount( ) obtains the current contour path motion that an axis is performing.

```
long int MCGetContouringCount(
   HCTRLR hCtlr,                  // controller handle
   WORD axis                      // axis number
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.

axis Axis number to query.

Returns
The return value is the number of linear or user defined contour path motions that have been completed.

Comments
This function allows the application to determine in what area of a continuous path motion an axis is at any given time.

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility
The MCAPI does not does not support contouring on the DC2, DCX-PC100, or DCX-PCI100 controllers.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes
Delphi: function MCGetContouringCount(hCtlr: HCTRLR; axis: Word): Longint; stdcall;
VB: Function MCGetContouringCount(ByVal hCtrlr As Integer, ByVal axis As Integer) As Long
LabVIEW: Not Supported

MCCL Reference
TX

See Also
MCGetContourConfig( ), **MCSetContourConfig( )**, **MCCONTOUR** structure definition

MCGetCount

MCGetCount( ) retrieves various count values from the specified *axis.*

```
long int MCGetCount(
   HCTRLR hCtlr,                 // controller handle
   WORD axis,                    // axis number
   long int type,                // type of count to retrieve
   long int* pCount              // variable to hold count value
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number to query.
type Specifies the type of data to retrieve:

Value	Description
MC_COUNT_CAPTURE	Retrieves the number of captured positions in high-speed capture mode.
MC_COUNT_COMPARE	Retrieves the number of successful comparisons in high-speed compare mode.
MC_COUNT_CONTOUR	Retrieves the index of the currently executing contour move in contouring mode.
MC_COUNT_FILTER	Retrieves the number of digital filter coefficients currently loaded.
MC_COUNT_FILTERMAX	Retrieves the maximum number of digital filter coefficients supported.

pCount Variable to hold requested count value.

Returns
MCERR_NOERROR is returned if there were no errors. If there was an error, one of the MCERR_xxxx error codes is returned.

Comments
MCGetCount( ) is a general purpose function for retrieving values related to high-speed capture mode, high-speed compare mode, contouring mode, and digital filter mode.

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility
The DC2 stepper axes, and the MC100, MC110, MC150, MC160 modules when installed on the DCX-PC100 controller do not support data capture. The DCX-PCI100 controller does not support torque mode nor do any stepper axes, which prevents the capture of torque values. The DC2, DCX-PC100, DCX-AT200, and DCX-PCI100 controllers do not support high-speed position compare. The MCAPI does not does not support contouring on the DC2, DCX-PC100, and DCX-PCI100 controllers. The DC2, DCX-PC100, DCX-AT200, DCX-PCI100 controllers, MC360, and MC362 modules do not support digital filtering.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 3.1 or higher

Prototypes
Delphi: function MCGetCount(hCtlr: HCTRLR; axis: Word; type: Longint; var pCount: Longint): Longint; stdcall;
VB: Function MCGetCount(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal argtype As Long, count As Long) As Long
LabVIEW: Not Supported

MCCL Reference
CG, GC, TX

See Also
MCGetContouringCount( )

MCGetDecelerationEx

MCGetDecelerationEx( ) returns the current programmed deceleration value for the given axis, in whatever units the axis is configured for.

```
long int MCGetDecelerationEx(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double* pDecel                 // deceleration return value
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to query.
pDecel	Pointer to a double precision floating point variable that will hold the deceleration for the specified axis.

Returns

The deceleration is placed in the variable specified by the pointer *pDecel* and MCERR_NOERROR is returned if there were no errors. If there was an error, one of the MCERR_xxxx error codes is returned and the variable pointed to by *pDecel* is left unchanged.

Comments

The deceleration value is the same as that reported by the **MCGetMotionConfigEx( )** function, these functions provide a short-hand method for obtaining just the deceleration value.

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility

There are no compatibility issues with this function.

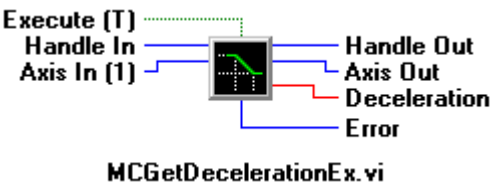
Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes

Delphi: function MCGetDecelerationEx(hCtlr: HCTRLR; axis: Word; var pDecel: Double): Longint; stdcall;
VB: Function MCGetDecelerationEx(ByVal hCtrlr As Integer, ByVal axis As Integer, decel As Double) As Long

LabVIEW:



MCCL Reference
Controller RAM Motor Tables

See Also
MCSetDeceleration( ), **MCGetMotionConfigEx( )**

MCGetDigitalFilter

MCGetDigitalFilter( ) obtains the digital filter coefficients for the specified axis.

```
long int MCGetDigitalFilter(
   HCTRLR hCtlr                  // controller handle
   WORD axis,                    // axis number
   double* pCoeff,               // array to hold retrieved coefficients
   long int num,                 // number of coefficients to retrieve
   long int* pActual             // number of valid coefficients retrieved
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to query.
pCoeff	Array to hold retrieved coefficients, must be *num* elements long (or longer). If this pointer is NULL, no coefficients are retrieved.
num	Number of coefficients to retrieve, cannot be larger than the maximum digital filter size supported by the controller.
pActual	Points to long integer that will be set equal to the number of valid coefficients currently loaded for this axis. If this pointer is NULL, no value is returned.

Returns
MCERR_NOERROR is returned if there were no errors. If there was an error, one of the MCERR_xxxx error codes is returned.

Comments
This function retrieves zero or more of the digital filter coefficients currently loaded in an axis. Optionally the actual number of loaded coefficients is also returned (this value is also available from **MCGetCount( )**).

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility
The DC2, DCX-PC100, DCX-AT200, DCX-PCI100 controllers, MC360, and MC362 modules do not support digital filtering.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 3.1 or higher

Prototypes
Delphi: function MCGetDigitalFilter(hCtlr: HCTRLR; axis: Word; coeff: Array of Double; num: Longint; var pActual: Longint): Longint; stdcall;
VB: Function MCGetDigitalFilter(ByVal hCtrlr As Integer, ByVal axis As Integer, coeff As Double, ByVal num As Long, actual As Long) As Long
LabVIEW: Not Supported

MCCL Reference
GF

See Also
MCEnableDigitalFilter( ), **MCGetCount( )**, **MCIsDigitalFilter( )**, **MCSetDigitalFilter( )**

MCGetError

MCGetError( ) returns the most recent error code for *hCtlr*.

```
short int MCGetError(
   HCTRLR hCtlr                  // controller handle
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.

Returns
The return value is a numeric error code (or MCERR_NOERROR if there is no error) for the most recent error detected for the specified controller.

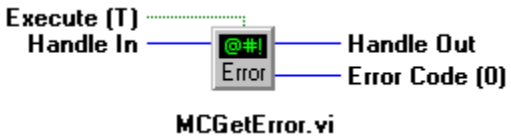
Comments
The error is cleared (set equal to MCERR_NOERROR) after it has been read. Errors are maintained on a per-handle basis, such that calls to **MCGetError( )** only return errors that occurred during function calls that used the same handle.

A more flexible way to detect errors is to use the **MCErrorNotify( )**. This function delivers error messages directly to the window procedure of your choice.

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.2 or higher

Prototypes
Delphi: function MCGetError(hCtlr: HCTRLR): SmallInt; stdcall;
VB: Function MCGetError(ByVal hCtrlr As Integer) As Integer
LabVIEW:



MCCL Reference
None

See Also
MCErrorNotify( ), **MCTranslateErrorEx( )**

MCGetFilterConfigEx

MCGetFilterConfigEx( ) obtains the current PID filter configuration for a servo motor or the closed-loop configuration for a stepper motor operating in closed-loop mode. Please see the online MCAPI Reference for the **MCGetFilterConfig( )** prototype.

```
long int MCGetFilterConfigEx(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   MCFILTEREX* pFilter            // address of filter configuration
                                  // structure
);
```

Parameters
hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to query.
pFilter	Points to an **MCFILTEREX** structure that receives the PID filter configuration information for *axis*.

Returns
MCGetFilterConfigEx( ) places the PID filter settings in the structure specified by the pointer *pFilter*. MCERR_NOERROR is returned if there were no errors. If there was an error, one of the MCERR_xxxx error codes is returned.

Comments
This function must be used to obtain the current PID filter configuration for a servo motor or the closed-loop configuration for a stepper motor operating in closed-loop mode.

Closed-loop stepper operation requires firmware version 2.1a or higher on the DCX-PCI300 and firmware version 2.5a or higher on the DCX-AT300.

You may not set the *axis* parameter to MC_ALL_AXES for this command..

Compatibility

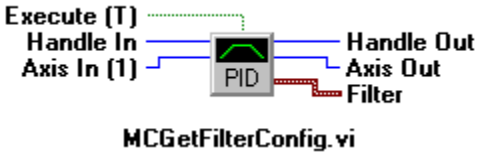
VelocityGain is not supported on the DCX-PCI100 controller, MC100, MC110 modules, or closed-loop steppers. **AccelGain** is not supported on the DC2, DCX-PC100, and DCX-PCI100 controllers. **DecelGain** is not supported on the DC2, DCX-PC100, and DCX-PCI100 controllers.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 3.2 or higher

Prototypes

Delphi: function MCGetFilterConfigEx(hCtlr: HCTRLR; axis: Word; var pFilter: MCFILTEREX): SmallInt; stdcall;
VB: Function MCGetFilterConfigEx(ByVal hCtrlr As Integer, ByVal axis As Integer, filter As MCFilterEx) As Integer
LabVIEW:



MCCL Reference

TD, TF, TG, TI, TL, Controller RAM Motor Tables

See Also

MCSetFilterConfigEx( ), **MCFILTEREX** structure definition

MCGetFollowingError

MCGetFollowingError( ) returns the current following error (difference between the actual and the optimal positions) for the specified axis.

```
long int MCGetFollowingError(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double* pError                 // following error return value
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to query.
pError	Points to a double precision variable that will hold the following error.

Returns
This function places the following error in the variable specified by the pointer *pError*, and MCERR_NOERROR is returned if there were no errors. If there was an error, one of the MCERR_xxxx error codes is returned and the variable pointed to by pError is left unchanged.

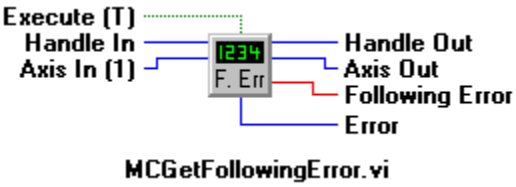
Comments

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes
Delphi: function MCGetFollowingError(hCtlr: HCTRLR; axis: Word; var pError: Double): Longint; stdcall;
VB: Function MCGetFollowingError(ByVal hCtrlr As Integer, ByVal axis As Integer, error As Double) As Long
LabVIEW:



MCCL Reference
TF

See Also
MCGetOptimalEx( ), **MCGetPositionEx( )**

MCGetGain

MCGetGain( ) returns the current gain setting for the specified axis.

```
long int MCGetGain(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double* pGain                  // gain return value
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to query.
pGain	Points to a double precision variable that will hold the gain value.

Returns

MCGetGain( ) places the gain value in the variable specified by the pointer *pGain* and MCERR_NOERROR is returned if there were no errors. If there was an error, one of the MCERR_xxxx error codes is returned and the variable pointed to by *pGain* is left unchanged.

Comments

The gain value is the same as that reported by the **MCGetMotionConfigEx( )** function, this function provide a short-hand method for obtaining just the gain value.

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility

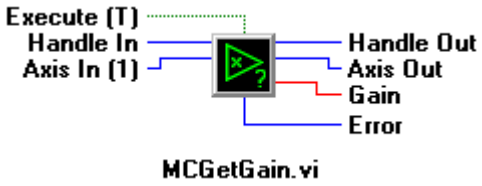
There are no compatibility issues with this function.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes

Delphi: function MCGetGain(hCtlr: HCTRLR; axis: Word; var pGain: Double): Longint; stdcall;
VB: Function MCGetGain(ByVal hCtrlr As Integer, ByVal axis As Integer, gain As Double) As Long
LabVIEW:



MCCL Reference

TG

See Also

MCGetMotionConfigEx( ) , **MCSetGain( )**

MCGetIndexEx

MCGetIndexEx( ) returns the position where the encoder index pulse was observed for the specified axis, in whatever units the axis is configured for.

```
long int MCGetIndexEx(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double* pIndex                 // index position return value
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to query.
pIndex	Pointer to a double precision floating point variable that will hold the index position for the specified axis.

Returns

The index position is placed in the variable specified by the pointer *pIndex* and MCERR_NOERROR is returned if there were no errors. If there was an error, one of the MCERR_xxxx error codes is returned and the variable pointed to by pIndex is left unchanged.

Comments

Controller resets and the **MCSetPosition( )** function may be change the position reading of the primary encoder.

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility

The MC100, MC110 modules, and all stepper axes do not support this function.

Requirements

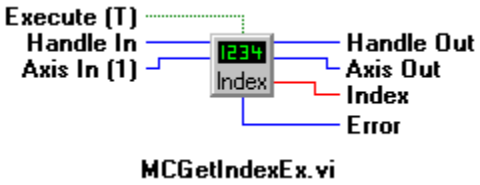
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes

Delphi: function MCGetIndexEx(hCtlr: HCTRLR; axis: Word; var pIndex: Double): Longint; stdcall;

VB: Function MCGetIndexEx(ByVal hCtrlr As Integer, ByVal axis As Integer, index As Double) As Long

LabVIEW:



MCCL Reference

TZ

See Also

MCGetAuxEncIdxEx( ), **MCSetPosition( )**

MCGetInstalledModules

MCGetInstalledModules( ) enumerates the types of modules installed on a motion controller.

```
long int MCGetInstalledModules(
   HCTRLR hCtlr,                 // controller handle
   long int* modules,            // pointer to an array for controller type
                                 // IDs
   long int size                 // size of Modules array
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
modules	Pointer to an array of long integers, filled with module types on return.
size	Size of *modules* array (number of integers).

Returns

MCERR_NOERROR is returned if there were no errors. If there was an error, one of the MCERR_xxxx error codes is returned.

Comments

MCGetInstalledModules( ) fills the *modules* array with module type identifiers, where the type of module installed in position #1 on the controller is stored in Modules[0], the type of module installed in position #2 on the controller is stored in Modules[1], etc. In order to list all installed controllers the array must have a size at least equal to the value in the **MaximumModules** field of the **MCPARAMEX( )** data structure.

Compatibility

There are no compatibility issues with this function.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 3.0 or higher

Prototypes

Delphi:	function MCGetInstalledModules(hCtlr: HCTRLR; modules: Array of LongInt; size: LongInt): Longint; stdcall;
VB:	Function MCGetInstalledModules(ByVal hCtrlr As Integer, modules As Any, ByVal size As Long) As Long
LabVIEW:	Not Supported

MCCL Reference

None

See Also

MCGetConfigurationEx( )

MCGetJogConfig

MCGetJogConfig( ) obtains the current jog configuration block for the specified axis.

```
short int MCGetJogConfig(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   MCJOG* pJog                    // address of jog configuration structure
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number from which to retrieve jog information.
pJog	Points to a **MCJOG** structure that contains jog configuration information for *axis*.

Returns

The return value is TRUE if the function is successful. Otherwise it returns FALSE, indicating the function did not find the *axis* specified (hCtlr or *axis* incorrect).

Comments

This function must be used to obtain current jog configuration information for an axis.

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility

The DCX-PCI controllers, DC2 stepper axes, MC150, and MC160 modules do not support jogging.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: function MCGetJogConfig(hCtlr: HCTRLR; axis: Word; var pJog: MCJOG): SmallInt; stdcall;
VB: Function MCGetJogConfig(ByVal hCtrlr As Integer, ByVal axis As Integer, jog As MCJog) As Integer
LabVIEW: Not Supported

MCCL Reference

Controller RAM Motor Tables

See Also

MCEnableJog( ), **MCGetJogConfig( )**, **MCJOG** structure definition

MCGetLimits

MCGetLimits( ) obtains the current hard and soft limit settings for the specified axis.

```
long int MCGetLimits(
   HCTRLR hCtlr,                // controller handle
   WORD axis,                   // axis number
   short int* pHardMode,        // hard limit mode flags
   short int* pSoftMode,        // soft limit mode flags
   double* pLimitMinus,         // soft low limit value
   double* pLimitPlus           // soft high limit value
);
```

Parameters

hCtlr Controller handle, returned by a successful call to **MCOpen( )**.

axis Axis number to query.

pHardMode Combination of limit mode flags for the hard limits. See description of *pSoftMode* for details.

pSoftMode Combination of the following limit mode flags for the soft limits:

Value	Description
MC_LIMIT_PLUS	Enables the positive limit.
MC_LIMIT_MINUS	Enables the negative limit.
MC_LIMIT_BOTH	Enables both the positive and negative limits.
MC_LIMIT_OFF	Limit stopping mode is set to turn the motor off when a limit is tripped.
MC_LIMIT_ABRUPT	Limit stopping mode is set to abrupt (target position is set to current position and PID loop stops axis as quickly as possible).
MC_LIMIT_SMOOTH	Limit stopping mode is set to smooth (axis executes pre-programmed deceleration when limit is tripped).
MC_LIMIT_INVERT	Inverts the polarity of the hardware limit switch inputs. This value may not be used with soft limits.

pLimitMinus Pointer to a variable where the negative limit value for soft limits, if supported by this controller, will be stored.

pLimitPlus Pointer to a variable where the positive limit value for soft limits, if supported by this controller, will be stored.

Returns

MCGetLimits( ) returns the value MCERR_NOERROR if the function completed without errors. If there was an error, one of the MCERR_xxxx error codes is returned, and the variables pointed to by the function pointers will be left in an undetermined state.

Comments

The limit settings are the same as those reported by the **MCGetMotionConfigEx( )** function, this function provide a short-hand method for obtaining just the limit settings.

Beginning with Version 2.23 of the Motion Control API you may pass a NULL pointer for *pHardMode*, *pSoftMode*, *pLimitMinus*, or *pLimitPlus*. This permits a program to easily ignore values it is not interested in. A program that needs to check the Hard Limit settings might set all the pointers for Soft Limit values to NULL to ignore those values, as opposed to

having to create dummy variables to hold the values that will never be used. Because this feature is new in Version 2.23, only applications that do not require backward compatibility with an earlier MCAPI version should take advantage of it.

You may not set the *axis* parameter to MC_ALL_AXES for this command.

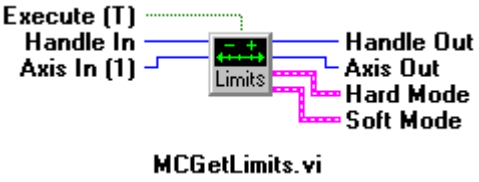
Compatibility
The DC2 and DCX-PC100 controllers do not support soft limits.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes
Delphi: function MCGetLimits(hCtlr: HCTRLR; axis: Word; var pHardMode, pSoftMode: SmallInt; var pLimitMinus, pLimitPlus: Double): Longint; stdcall;

VB: Function MCGetLimits(ByVal hCtrlr As Integer, ByVal axis As Integer, hardMode As Integer, softMode As Integer, limitMinus As Double, limitPlus As Double) As Long

LabVIEW:



MCCL Reference
Controller RAM Motor Tables

See Also
MCGetMotionConfigEx( ), **MCSetLimits( )**, **MCSetMotionConfigEx( )**

MCGetModuleInputMode

MCGetModuleInputMode( ) returns the current input mode for the specified axis.

```
long int MCGetModuleInputMode(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   long int* mode                 // input mode value
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number to query.

mode Pointer to a long integer variable that will hold the input mode for the specified axis:

Value	Description
MC_IM_OPENLOOP	Stepper motor axis is in open-loop mode.
MC_IM_CLOSEDLOOP	Stepper motor axis is in closed-loop mode.

Returns
The return value is MCERR_NOERROR if no errors were detected. If there was an error, one of the MCERR_xxxx error codes is returned and the variable pointed to by *mode* is left unchanged.

Comments

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility
The DC2, DCX-PC100, DCX-PCI100, DCX-AT100, and DCX-AT200 controllers do not support a module which is capable of closed-loop stepper operation. The MC362 module is not capable of closed-loop stepper operation.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 3.2 or higher

Prototypes
Delphi: function MCGetModuleInputMode(hCtlr: HCTRLR; axis: Word; var mode: LongInt): Longint; stdcall;
VB: Function MCGetModuleInputMode(ByVal hCtrlr As Integer, ByVal axis As Integer, mode As Long) As Long
LabVIEW: Not Supported

MCCL Reference
IM

See Also
MCSetModuleInputMode( )

MCGetMotionConfigEx

MCGetMotionConfigEx( ) obtains the current motion configuration block for the specified axis.

```
short int MCGetMotionConfigEx(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   MCMOTIONEX* pMotion            // address of motion configuration
                                  // structure
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to query.
pMotion	Points to an **MCMOTIONEX** structure that receives motion configuration information for *axis*.

Returns

The return value is TRUE if the function is successful. A return value of FALSE indicates the function did not find the *axis* specified (*hCtlr* or *axis* incorrect).

Comments

This function provides a way of initializing a **MCMOTIONEX** structure with the current motion parameters for the given *axis*. When you need to setup many of the parameters for an axis it is easier to call **MCGetMotionConfigEx( )**, update the **MCMOTIONEX** structure, and write the changes back using **MCSetMotionConfigEx( )**, rather than use a Get/Set function call for each parameter.

Note that some less often used parameters will only be accessible from this function and from **MCSetMotionConfigEx( )** - they do not have individual Get/Set functions.

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility

Acceleration is not supported on the DC2 stepper axes. **Deceleration** is not supported on the DCX-PCI100 controller, MC100, MC110, MC150, or MC160 modules. **MinVelocity** is not supported on the DCX-PCI100, DCX-PC100, or DC2 controllers. **Torque** is not supported on the DCX-PCI100 controller, MC100, or MC110 modules. **Deadband** is not supported on the DCX-PC100 controller, DC2 stepper axes, MC150, MC160, MC260, MC360 or MC362 modules. **DeadbandDelay** is not supported on the DCX-PC100 controller, DC2 stepper axes, MC150, MC160, MC260, MC360 or MC362 modules. **StepSize** is not supported on the DC2 or DCX-PCI100 controllers. **Current** is not supported on the DC2 or DCX-PCI100 controllers. **SoftLimitMode** is not supported on the DC2 or DCX-PC100 controllers. **SoftLimitLow** is not supported on the DC2 or DCX-PC100 controllers. **SoftLimitHigh** is not supported on the DC2 or DCX-PC100 controllers. **EnableAmpFault** is not supported on the DC2 controllers. **UpdateRate** is not supported on the DC2 or DCX-PCI100 controllers.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: function MCGetMotionConfigEx(hCtlr: HCTRLR; axis: Word; var pMotion: MCMOTIONEX): SmallInt; stdcall;
VB: Function MCGetMotionConfigEx(ByVal hCtrlr As Integer, ByVal axis As Integer, motion As MCMotionEx) As Integer
LabVIEW: Not Supported

MCCL Reference
TG, Controller RAM Motor Tables

See Also
MCSetMotionConfigEx( ), **MCMOTIONEX** structure definition

MCGetOperatingMode

MCGetOperatingMode( ) returns the current operating mode for the specified axis.

```
long int MCGetOperatingMode(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   long int* mode                 // operating mode value
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number to query.
mode Pointer to a long integer variable that will hold the operating mode for the specified axis:

Value	Description
MC_MODE_CONTOUR	Contouring mode operation.
MC_MODE_GAIN	Gain mode operation.
MC_MODE_POSITION	Position mode operation.
MC_MODE_TORQUE	Torque mode operation.
MC_MODE_UNKNOWN	Unable to determine current mode of operation.
MC_MODE_VELOCITY	Velocity mode operation.

Returns
The return value is MCERR_NOERROR if no errors were detected. If there was an error, one of the MCERR_xxxx error codes is returned and the variable pointed to by *mode* is left unchanged.

Comments

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas

Library: use mcapi32.lib
Version: MCAPI 3.2 or higher

Prototypes

Delphi: function MCGetOperatingMode(hCtlr: HCTRLR; axis: Word; var mode: LongInt): Longint; stdcall;
VB: Function MCGetOperatingMode(ByVal hCtrlr As Integer, ByVal axis As Integer, mode As Long) As Long
LabVIEW: Not Supported

MCCL Reference

None

See Also

MCSetOperatingMode( )

MCGetOptimalEx

MCGetOptimalEx( ) returns the current optimal position from the trajectory generator for the specified axis, in whatever units the axis is configured for.

```
long int MCGetOptimalEx(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double* pOptimal               // optimal return value
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to query.
pOptimal	Pointer to a double precision floating point variable that will hold the optimal position for the specified axis.

Returns

The optimal position is placed in the variable specified by the pointer *pOptimal* and a zero is returned, if there were no errors. If there was an error, one of the MCERR_xxxx error codes is returned and the variable pointed to by *pOptimal* is left unchanged.

Comments

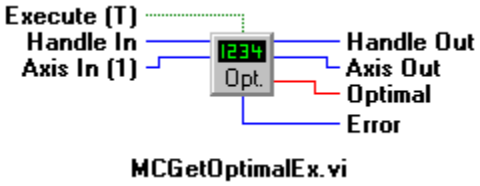
The trajectory generator generates an optimal position based upon an ideal (i.e. error free) motor. The PID loop then compares the actual position to the optimal position to calculate a correction to the actual trajectory. The maximum difference allowed between the optimal and actual positions is set with the **FollowingError** member of an **MCFILTEREX** structure.

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility
The DC2 stepper axes do not support this command.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes
Delphi: function MCGetOptimalEx(hCtlr: HCTRLR; axis: Word; var pOptimal: Double): Longint; stdcall;
VB: Function MCGetOptimalEx(ByVal hCtrlr As Integer, ByVal axis As Integer, optimal As Double) As Long
LabVIEW:



MCCL Reference
TO

See Also
MCGetFilterConfigEx( ), **MCSetFilterConfigEx( )**, **MCSetPosition( )**

MCGetPositionEx

MCGetPositionEx( ) returns the current position for the specified axis, in whatever units the axis is configured for.

```
void MCGetPositionEx(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double* pPosition              // position return value
);
```

Parameters
hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to query.
pPosition	Pointer to a double precision floating point variable that will hold the position for the specified axis.

Returns
The position value is placed in the variable specified by the pointer *pPosition* and a zero is returned, if there were no errors. If there was an error, one of the MCERR_xxxx error codes is returned and the variable pointed to by *pPosition* is left unchanged.

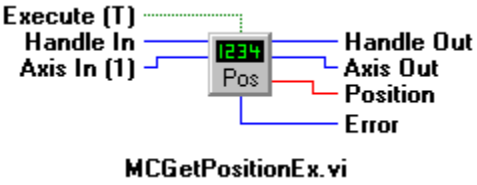
Comments

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes
Delphi: function MCGetPositionEx(hCtlr: HCTRLR; axis: Word; var pPosition: Double): Longint; stdcall;
VB: Function MCGetPositionEx(ByVal hCtrlr As Integer, ByVal axis As Integer, position As Double) As Long
LabVIEW:



MCCL Reference
TP

See Also
MCSetPosition( ), **MCSetScale( )**

MCGetProfile

MCGetProfile( ) returns the current acceleration / deceleration profile for the specified axis.

```
long int MCGetProfile(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   WORD* pProfile                 // profile return value
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number to query.
pProfile Pointer to a WORD variable that will hold the profile for the specified axis:

Value	Description
MC_PROF_PARABOLIC	Indicates that a parabolic acceleration / deceleration profile has been selected.

Value	Description
MC_PROF_SCURVE	Indicates that an S-curve acceleration / deceleration profile has been selected.
MC_PROF_TRAPEZOID	Indicates that a trapezoidal acceleration / deceleration profile has been selected.
MC_PROF_UNKNOWN	This value is returned when **MCGetProfile( )** cannot determine the current profile setting.

Returns
The return value is MCERR_NOERROR, if no errors were detected. If there was an error, the return value is one of the MCERR_xxxx error codes is returned and the variable pointed to by *pProfile* is left unchanged.

Comments
To determine if the controller supports user configurable acceleration profiles check the **CanChangeProfile** field of the **MCPARAMEX** structure returned by **MCGetConfigurationEx( )**.

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes
Delphi: function MCGetProfile(hCtlr: HCTRLR; axis: Word; var pProfile: Word): Longint; stdcall;
VB: Function MCGetProfile(ByVal hCtrlr As Integer, ByVal axis As Integer, profile As Integer) As Long
LabVIEW: Not Supported

MCCL Reference
Controller RAM Motor Tables

See Also
MCSetProfile( ), **MCPARAMEX** structure definition

MCGetRegister

MCGetRegister( ) returns the value of the specified general purpose register.

```
long int MCGetRegister(
   HCTRLR hCtlr,                  // controller handle
   long int register,             // register number
   void* pValue                   // pointer to variable to hold register
                                  // value
   long int type                  // type of variable pointed to by pValue
);
```

Parameters

hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
register Register number to read from (0 to 255).
pValue Pointer to a variable that will hold the register contents.
type Type of data pointed to by *pValue*:

Value	Description
MC_TYPE_LONG	Indicates *pValue* points to a variable of type long integer.
MC_TYPE_DOUBLE	Indicates *pValue* points to a variable of type double precision floating point.
MC_TYPE_FLOAT	Indicates *pValue* points to a variable of type single precision floating point.

Returns

The return value is MCERR_NOERROR, if no errors were detected. If there was an error, the return value is one of the MCERR_xxxx error codes is returned and the variable pointed to by *pValue* is left unchanged.

Comments

MCGetRegister( ) and **MCSetRegister( )** allow you to read from and write to, respectively, the general purpose registers on the motion controller. When running background tasks on a multitasking controller the only way to communicate with the background tasks is to pass parameters in the general purpose registers.

You cannot read from the local registers (registers 0 - 9) of a background task. When you need to communicate with a background task be sure to use one or more of the global registers (10 - 255).

To determine if your controller supports multi-tasking check the **MultiTasking** field of the **MCPARAMEX** structure returned by **MCGetConfigurationEx( )**.

Compatibility

There are no compatibility issues with this function.

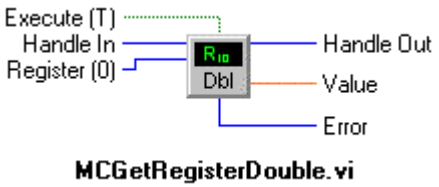
Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 2.0 or higher

Prototypes

Delphi: function MCGetRegister(hCtlr: HCTRLR; register: Longint; var pValue: Pointer; type: Longint): Longint; stdcall;
VB: Function MCGetRegister(ByVal hCtrlr As Integer, ByVal register As Long, value As Any, ByVal argtype As Long) As Long

LabVIEW:



MCCL Reference
TR

See Also
MCSetRegister( )

MCGetScale

MCGetScale( ) obtains the current scaling factors for the specified axis.

```
void MCGetScale(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   MCSCALE* pScale                // address of scale factors structure
);
```

Parameters
hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to query.
pScale	Pointer to a **MCSCALE** structure that will hold scaling information for *axis*.

Returns
The return value is TRUE if the function is successful. A return value of FALSE indicates the function did not find the *axis* specified (*hCtlr* or *axis* incorrect).

Comments
Scaling allows the application to communicate with the controller in real world units such as inches, meters, and radians; as opposed to low level (i.e. un-scaled) values such as raw encoder counts, etc.

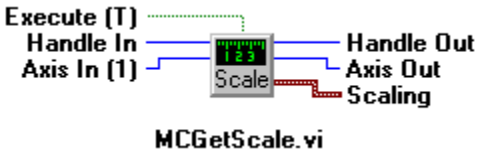
In order to see if a controller supports scaling, an application can test the Boolean flag **CanDoScaling** in the **MCPARAMEX** structure returned by **MCGetConfigurationEx( )**.

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility
The DC2 and DCX-PC controllers do not support scaling.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes
Delphi: function MCGetScale(hCtlr: HCTRLR; axis: Word; var pScale: MCSCALE): SmallInt; stdcall;
VB: Function MCGetScale(ByVal hCtrlr As Integer, ByVal axis As Integer, scale As MCScale) As Integer
LabVIEW:



MCGetScale.vi

MCCL Reference
Controller RAM Motor Tables

See Also
MCGetConfigurationEx( ), **MCSetScale( )**, **MCSCALE** structure definition

MCGetServoOutputPhase

MCGetServoOutputPhase( ) returns the current servo output phasing for the specified axis.

```
long int MCGetServoOutputPhase(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   WORD* pPhase                   // phase return value
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number to query for phase setting.
pPhase Pointer to a WORD variable that will hold the phase setting for the specified axis:

Value	Description
MC_PHASE_STD	Indicates that the axis is configured for standard phasing.
MC_PHASE_REV	Indicates that the axis is configured for reverse phasing.

Returns
The return value is MCERR_NOERROR if no errors were detected. If there was an error, the return value is one of the MCERR_xxxx error codes is returned, and the variable pointed to by *pPhase* is left unchanged.

Comments

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility
The MC100 and MC110 modules do not support phase reverse.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes
Delphi: function MCGetServoOutputPhase(hCtlr: HCTRLR; axis: Word; var pPhase: Word): Longint; stdcall;
VB: Function MCGetServoOutputPhase(ByVal hCtrlr As Integer, ByVal axis As Integer, phase As Integer) As Long
LabVIEW: Not Supported

MCCL Reference
None

See Also
MCSetServoOutputPhase( )

MCGetStatus

MCGetStatus( ) returns the controller dependent status word for the specified axis.

```
long int MCGetStatus(
   HCTRLR hCtlr,                  // controller handle
   WORD axis                      // axis number
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number to query.

Returns
The return value is the 32-bit status word for the selected axis.

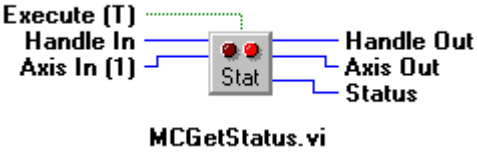
Comments
Please refer to the hardware manual for your controller for specific information about meaning and location of the flags located within the status word. As an alternative, the MCAPI function **MCDecodeStatus( )** provides a controller-independent way to process the flags in the status word.

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes
Delphi: function MCGetStatus(hCtlr: HCTRLR; axis: Word): Longint; stdcall;
VB: Function MCGetStatus(ByVal hCtrlr As Integer, ByVal axis As Integer) As Long
LabVIEW:



MCCL Reference
TS

See Also
MCDecodeStatus( ), Controller hardware reference manual

MCGetTargetEx

MCGetTargetEx( ) returns the move target position, as set by the most recent **MCMoveAbsolute( )** or **MCMoveRelative( )** function call, for the specified axis.

```
void MCGetTargetEx(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double* pTarget                // target position return
);
```

Parameters
hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to query.
pTarget	Pointer to a double precision floating point variable that will hold the target position for the specified axis.

Returns
The target position value is placed in the variable specified by the pointer *pTarget* and MCERR_NOERROR is returned if there were no errors. If there was an error, one of the MCERR_xxxx error codes is returned, and the variable pointed to by *pTarget* is left unchanged.

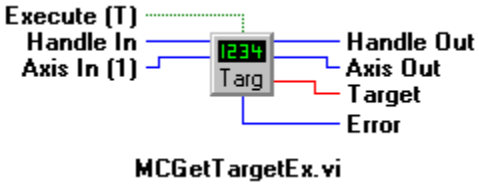
Comments
The API move functions **MCMoveAbsolute( )** and **MCMoveRelative( )** update the target position for an axis. The controller will then generate an optimal trajectory to the target position, and the PID loop will seek to minimize the error (difference between actual and optimal trajectories).

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes
Delphi: function MCGetTargetEx(hCtlr: HCTRLR; axis: Word; var pTarget: Double): Longint; stdcall;
VB: Function MCGetTargetEx(ByVal hCtrlr As Integer, ByVal axis As Integer, target As Double) As Long
LabVIEW:



MCCL Reference
TT

See Also
MCMoveAbsolute( ), **MCMoveRelative( )**

MCGetTorque

MCGetTorque( ) returns the current torque setting for the specified axis.

```
long int MCGetTorque(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double* pTorque                // torque return value
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to query.
pTorque	Points to a double precision variable that will hold the torque.

Returns

MCGetTorque( ) places the torque setting in the variable specified by the pointer *pTorque* and a zero is returned if there were no errors. If there was an error, one of the MCERR_xxxx error codes is returned, and the variable pointed to by *pTorque* is left unchanged.

Comments

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility

Torque mode is not supported on stepper axes, DCX-PCI100 controller, MC100, or MC110 modules.

Requirements

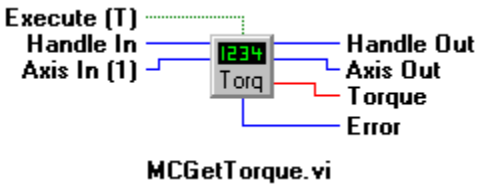
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes

Delphi: function MCGetTorque(hCtlr: HCTRLR; axis: Word; var pTorque: Double): Longint; stdcall;

VB: Function MCGetTorque(ByVal hCtrlr As Integer, ByVal axis As Integer, torque As Double) As Long

LabVIEW:



MCCL Reference

TQ

See Also

MCGetMotionConfigEx( ), **MCSetMotionConfigEx( )**, **MCSetTorque( )**, **MCMOTIONEX** structure definition

MCGetVectorVelocity

MCGetVectorVelocity( ) returns the current programmed velocity for the specified axis, in whatever units the axis is configured for.

```
long int MCGetVectorVelocity(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double* pVelocity              // vector velocity return value
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to query.
pVelocity	Pointer to a double precision floating point variable that will hold the vector velocity value for the specified axis.

Returns

The position value is placed in the variable specified by the pointer *pVelocity* and MCERR_NOERROR is returned if there were no errors. If there was an error, one of the MCERR_xxxx error codes is returned, and the variable pointed to by *pVelocity* is left unchanged.

Comments

The vector velocity value for a particular *axis* may also be obtained using **MCGetContourConfig( )**.
MCGetVectorVelocity( ) provides a short-hand method for getting just the vector velocity value and is most useful when updating vector velocity settings on the fly.

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility

The MCAPI does not does not support contouring on the DC2, DCX-PC100, or DCX-PCI100 controllers.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 2.0 or higher

Prototypes

Delphi:	function MCGetVectorVelocity(hCtlr: HCTRLR; axis: Word; var pVelocity: Double): Longint; stdcall;
VB:	Function MCGetVectorVelocity(ByVal hCtrlr As Integer, ByVal axis As Integer, velocity As Double) As Long
LabVIEW:	Not Supported

MCCL Reference

None

See Also

MCGetContourConfig( ), **MCSetVectorVelocity( )**

MCGetVelocityEx

MCGetVelocityEx( ) returns the current programmed velocity for the specified axis, in whatever units the axis is configured for.

```
long int MCGetVelocityEx(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // axis number
   double* pVelocity              // velocity return value
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to query.
pVelocity	Pointer to a double precision floating point variable that will hold the velocity value for the specified axis.

Returns

The position value is placed in the variable specified by the pointer *pVelocity*, and MCERR_NOERROR is returned if there were no errors. If there was an error, one of the MCERR_xxxx error codes is returned, and the variable pointed to by *pVelocity* is left unchanged.

Comments

The programmed velocity value for a particular *axis* may also be obtained using the **MCGetMotionConfigEx( )** function. **MCGetVelocityEx( )** provides a short-hand method for getting just the velocity value and is most useful when updating velocity settings on the fly in velocity mode.

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility

There are no compatibility issues with this function.

Requirements

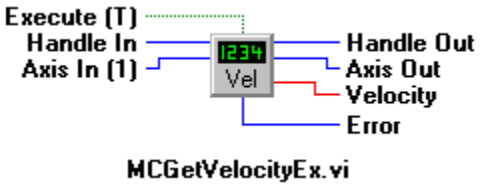
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes

Delphi: function MCGetVelocityEx(hCtlr: HCTRLR; axis: Word; var pVelocity: Double): Longint; stdcall;

VB: Function MCGetVelocityEx(ByVal hCtrlr As Integer, ByVal axis As Integer, velocity As Double) As Long

LabVIEW:



MCCL Reference
Controller RAM Motor Tables

See Also
MCSetVelocity( ), **MCSetMotionConfigEx( )**

MCIsAtTarget

MCIsAtTarget( ) waits for the "At Target" condition to go true for the specified axis. Use it to determine when motion has completed for an axis.

```
long int MCIsAtTarget(
   HCTRLR hCtlr,                // controller handle
   WORD axis,                   // axis number
   double timeout               // timeout, in seconds
);
```

Parameters
hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number for which to wait for the "At Target" condition.
timeout	Time to wait, in seconds, for the At Target condition to go true.

Returns
This function returns TRUE, if the axis is "At Target." A return value of FALSE indicates the specified axis is not "At Target" by the end of *timeout*. If MC_ALL_AXES is specified for Axis, TRUE will be returned only if all axes are "At Target."

Comments
This function waits for up to *timeout* seconds for the At Target status of the axis to be TRUE. It returns as soon as the status goes TRUE or when *timeout* expires. Set *timeout* to zero to check the At Target status only once and return immediately (i.e. no wait is performed).

Compatibility
The DC2, DCX-PC, and DCX-PCI100 do not support the At Target status bit and should use **MCIsStopped( )** instead.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 2.2 or higher

Prototypes
Delphi: function MCIsAtTarget(hCtlr: HCTRLR; axis: Word; timeout: Double): Longint; stdcall;
VB: Function MCIsAtTarget(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal timeout As Double) As Long
LabVIEW: Not Supported

MCCL Reference
None

See Also
MCIsStopped( )

MCIsDigitalFilter

MCIsDigitalFilter( ) is used to determine the enabled state of the digital filter mode.

```
long int MCIsDigitalFilter(
   HCTRLR hCtlr,                  // controller handle
   WORD axis                      // axis number
);
```

Parameters
hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number to query.

Returns
This function returns TRUE if the digital filter for the specified axis is enabled, or it returns FALSE if the digital filter is disabled.

Comments
This function is used to determine the enabled state of the digital filter mode supported by advanced motion control modules, such as the MC300.

You may not set the *axis* parameter to MC_ALL_AXES for this command.

Compatibility
The DC2, DCX-PC100, DCX-AT200, DCX-PCI100 controllers, MC360 and MC362 modules do not support digital filtering.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 3.1 or higher

Prototypes
Delphi:	function MCIsDigitalFilter(hCtlr: HCTRLR; axis: Word): Longint; stdcall;
VB:	Function MCIsDigitalFilter(ByVal hCtrlr As Integer, ByVal axis As Integer) As Long
LabVIEW:	Not Supported

MCCL Reference
None

See Also
MCEnableDigitalFilter( ), **MCGetCount( )**, **MCGetDigitalFilter( )**, **MCSetDigitalFilter( )**

MCIsEdgeFound

MCIsEdgeFound( ) waits for the "Edge Found" condition to go true for the specified axis. Use it to determine when an open-loop stepper motor homing sequence has detected the edge sensor.

```
long int MCIsEdgeFound(
   HCTRLR hCtlr,                // controller handle
   WORD axis                    // axis number
   double timeout               // timeout, in seconds
);
```

Parameters
hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number for which to wait for the "Edge Found" condition.
timeout	Time to wait, in seconds, for the “Edge Found” condition to go true.

Returns
This function returns TRUE if the stepper axis has detected the edge input or FALSE if the axis has not detected the edge input by the end of *timeout*.

Comments
This function waits for up to *timeout* seconds for the Edge Found status of a stepper motor axis to go TRUE. It returns as soon as the status goes TRUE or when *timeout* expires. Set *timeout* to zero to check the edge found status only once and return immediately (i.e. no wait is performed). This function uses **MCDecodeStatus( )** internally to test the MC_STAT_EDGE_FOUND status bit.

Compatibility
The DC2, DCX-PC100, and DCX-AT200 controllers do not support this function. Stepper modules when run in closed-loop mode do not support this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 3.2 or higher

Prototypes
Delphi: function MCIsEdgeFound(hCtlr: HCTRLR; axis: Word; timeout: Double): Longint; stdcall;
VB: Function MCIsEdgeFound(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal timeout As Double) As Long
LabVIEW: Not Supported

MCCL Reference
TS

See Also
MCDecodeStatus( ), **MCEdgeArm( )**, **MCWaitForEdge( )**

MCIsIndexFound

MCIsIndexFound( ) waits for the "Index Found" condition to go true for the specified axis. Use it to determine when a servo or closed-loop stepper motor homing sequence has detected the encoder index.

```
long int MCIsIndexFound(
   HCTRLR hCtlr,                // controller handle
   WORD axis                    // axis number
   double timeout               // timeout, in seconds
);
```

Parameters
hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number for which to wait for the "Index Found" condition.
timeout	Time to wait, in seconds, for the “Index Found” condition to go true.

Returns
This function returns TRUE if the servo axis has detected the encoder index or FALSE if the axis has not detected the encoder index by the end of *timeout*.

Comments
This function waits for up to *timeout* seconds for the Index Found status of a servo motor axis to go TRUE. It returns as soon as the status goes TRUE or when *timeout* expires. Set *timeout* to zero to check the encoder index status only once and return immediately (i.e. no wait is performed). This function uses **MCDecodeStatus( )** internally to test the MC_STAT_INDEX_FOUND status bit.

Compatibility
The DC2, DCX-PC100, and DCX-AT200 controllers do not support this function. Stepper modules when run in open-loop mode with an auxiliary encoder do not support primary encoder functions such as this.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 3.2 or higher

Prototypes
Delphi: function MCIsIndexFound(hCtlr: HCTRLR; axis: Word; timeout: Double): Longint; stdcall;
VB: Function MCIsIndexFound(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal timeout As Double) As Long
LabVIEW: Not Supported

MCCL Reference
TS

See Also
MCDecodeStatus( ), **MCIndexArm( )**, **MCWaitForIndex( )**

MCIsStopped

MCIsStopped( ) waits for the "Trajectory Complete" condition to go true for the specified axis. Use it to determine when motion has completed for an axis.

```
long int MCIsStopped(
   HCTRLR hCtlr,                // controller handle
   WORD axis,                   // axis number
   double timeout               // timeout, in seconds
);
```

Parameters
hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number for which to wait for the "Trajectory Complete" condition.
timeout	Time to wait, in seconds, for the Trajectory Complete condition to go true.

Returns
This function returns TRUE if the axis is "Trajectory Complete." A return value of FALSE indicates the specified axis is not "Trajectory Complete" by the end of *timeout*. If MC_ALL_AXES is specified for Axis, TRUE will be returned only if all axes are "Trajectory Complete."

Comments
This function waits for up to *timeout* seconds for the Trajectory Complete status of the axis to be TRUE. It returns as soon as the status goes TRUE or when *timeout* expires. Set *timeout* to zero to check the Trajectory Complete status only once and return immediately (i.e. no wait is performed).

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 2.2 or higher

Prototypes
Delphi:	function MCIsStopped(hCtlr: HCTRLR; axis: Word; timeout: Double): Longint; stdcall;
VB:	Function MCIsStopped(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal timeout As Double) As Long
LabVIEW:	Not Supported

MCCL Reference
None

See Also
MCIsAtTarget( )

MCTranslateErrorEx

MCTranslateErrorEx( ) translates numeric error codes into ASCII text messages.

```
long int MCTranslateErrorEx(
   short int error,              // error code to translate
   char* buffer,                 // character buffer for message
   long int length               // length of Buffer, in bytes
);
```

Parameters
error	Numeric error code to translate.
buffer	String buffer to hold ASCII error message.
length	Length of string buffer (in bytes).

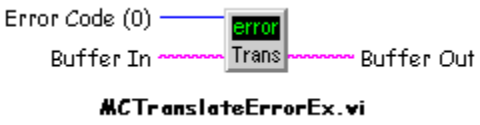
Returns
This function returns a pointer to the ASCII error message corresponding to Error. If Error does not correspond to a valid error message, a NULL pointer is returned. It will work with errors returned from **MCGetError( )** and **MCErrorNotify( )** error messages.

Comments
Beginning with version 2.1 of the MCAPI this function is included as a native MCAPI function (previously it was contained in a separate module). Incorporating **MCTranslateErrorEx( )** into the MCAPI DLL will facilitate future updates, but has required changes from how It previously worked. The string buffer and buffer length have been added to the argument list. These changes make it possible to call **MCTranslateErrorEx( )** from a much wider range of programming languages.

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 2.1 or higher

Prototypes
Delphi: function MCTranslateErrorEx(error: SmallInt; buffer: PChar; length: Longint): Longint; stdcall;
VB: Function MCTranslateErrorEx(ByVal error As Integer, ByVal buffer As String, ByVal length As Long) As Long
LabVIEW:



MCCL Reference

None

See Also

MCErrorNotify( ), **MCGetError( )**

Chapter Contents

Chapter 15

MCAPI I/O Functions

Digital I/O functions allow configuration of high or low “true” states, reading of inputs, sequencing based on input, and setting outputs. Analog I/O functions control the input and output of analog values through A/D and D/A ports installed on the controller.

A word of caution must be given regarding the use of board-level sequencing commands. Even though a warning is included with **MCWaitForDigitalIO( ),** it should be stressed that once this command is called, the board will not accept another command nor will it respond to the calling program until the board has completed what it was initially told to do. This can lead to scenarios where the calling program has absolutely no control during potentially dangerous or otherwise expensive situations.

To see examples of how the functions in this chapter are used, please refer to the online Motion Control API Reference.

MCConfigureDigitalIO

MCConfigureDigitalIO( ) configures a specific digital I/O channel for input or output and for high or low true logic.

```
short int MCConfigureDigitalIO(
   HCTRLR hCtlr,                // controller handle
   WORD channel,                // channel number
   WORD mode                    // configuration flags
);
```

Parameters

hCtlr Controller handle, returned by a successful call to **MCOpen( )**.

channel Digital channel number to configure.

mode Specifies how the channel is to be configured. This parameter may be any one of the digital I/O flags listed below. An input/output flag and a logic level flag may be OR'ed together.

Value	Description
MC_DIO_INPUT	Configures the channel for input.

Value	Description
MC_DIO_OUTPUT	Configures the channel for output.
MC_DIO_LOW	Configures the channel for negative logic level.
MC_DIO_HIGH	Configures the channel for positive logic level.
MC_DIO_LATCH	Configures the (input) channel for latched operation.

Returns

The return value is TRUE if the function is successful. A return value of FALSE indicates **MCConfigureDigitalIO( )** was unable to configure the channel as requested.

Comments

Each digital I/O channel may be configured for input or for output. The logic level maps the logical "on" and "off" states of the channel to the physical input and output voltages for that channel. If the channel is set to MC_DIO_LOW (negative logic) the "on" state of a channel will represent a low voltage (<0.4VDC) and "off" a high voltage (>2.4VDC). When set to MC_DIO_HIGH (positive logic) the "on" state of a channel will represent a high voltage (>2.4VDC) and "off" a low voltage (<.0.4VDC).

On the DC2-STN controller, beginning with firmware release 1.2a, it is possible to configure an input channel to "latch" input events (see the controller manual for details of signal hold time, etc.). Configure an input channel using the MC_DIO_LATCH constant to enable latching or clear the latched state. Configure an input channel using the MC_DIO_INPUT constant to disable latching.

The DCX-PCI motherboard has 16 general I/O, consisting of 8 fixed inputs and 8 fixed outputs. Since these digital I/O are fixed, they may not be configured for input or output. A program may verify the functionality (input or output) of a channel by using **MCGetDigitalIOConfig( )** to check the current configuration.

Under the MCAPI, the DC2-STN controller's input channels are numbered 1 - 8, and the output channels are numbered 9 - 16 (the MCAPI requires that each channel have a unique channel number).

Compatibility

There are no compatibility issues with this function.

Requirements

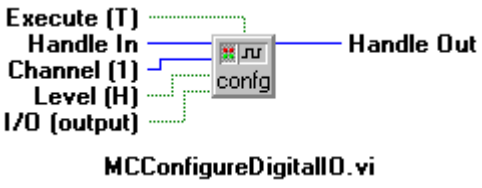
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: function MCConfigureDigitalIO(hCtlr: HCTRLR; channel, mode: Word): SmallInt;

VB: Function MCConfigureDigitalIO (ByVal hCtrlr As Integer, ByVal channel As Integer, ByVal mode As Integer) As Integer

LabVIEW:



MCCL Reference

CH, CI, CL, CT

See Also
MCEnableDigitalIO( ), **MCGetDigitalIO( )**, **MCGetDigitalIOConfig( )**

MCEnableDigitalIO

MCEnableDigitalIO( ) turns the specified digital I/O channel on or off.

```
void MCEnableDigitalIO(
   HCTRLR hCtlr,                // controller handle
   WORD channel,                // channel number
   short int state              // enable state
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
channel Digital channel number to enable.
state Specifies whether the channel is to be turned on or turned off.

Value	Description
TRUE	Turns the channel on.
FALSE	Turns the channel off.

Returns
This function does not return a value.

Comments
The I/O channel selected by *hCtlr* and *channel* must have previously been configured for output using the **MCConfigureDigitalIO( )** command. Note that depending upon how a channel has been configured "on" (and conversely "off") may represent either a high or a low voltage level.

state will accept any non-zero value as TRUE, and will work correctly with most programming languages, including those that define TRUE as a non-zero value other than one (one is the Windows default value for TRUE).

Under the MCAPI, the DC2-STN controller's input channels are numbered 1 - 8, and the output channels are numbered 9 - 16 (the MCAPI requires that each channel have a unique channel number).

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: mcapi32.lib

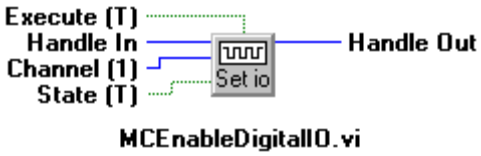
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure MCEnableDigitalIO(hCtlr: HCTRLR; channel: Word; state: SmallInt); stdcall;

VB: Sub MCEnableDigitalIO(ByVal hCtrlr As Integer, ByVal channel As Integer, ByVal state As Integer)

LabVIEW:



MCCL Reference

CF, CN

See Also

MCConfigureDigitalIO( ), **MCEnableDigitalIO( )**, **MCGetDigitalIOConfig( )**, **MCPARAMEX** structure definition

MCGetAnalog

MCGetAnalog( ) reads the current input state of the specified input Channel.

```
WORD MCGetAnalog(
   HCTRLR hCtlr,                  // controller handle
   WORD channel                   // channel number
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
channel	Analog channel number to read from.

Returns

This function returns the current A/D reading for *channel*.

Comments

The DC2, DCX-AT, and DCX-PC controllers all include four undedicated 8-bit analog input channels. By default these channels are assigned channel numbers 1 to 4. Each analog input accepts an input voltage between 0 and +5 volts. The value read in from the channel will be the ratio of the input voltage to the reference voltage times 255. An internal 5.0 volt reference is supplied by the controller; an external reference may be supplied in place of the internal reference if desired.

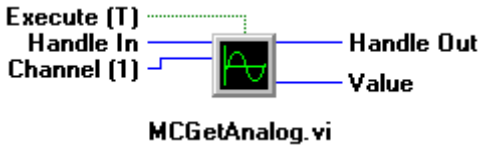
$$value = \frac{V_{Input}}{V_{Reference}} x\ 255$$

Additional analog input/output channels supplied by MC500 modules will occupy sequential channel numbers beginning with channel 5. The fields **AnalogInput** and **AnalogOutput** in the **MCPARAMEX** structure contain the number of input and output channels the controller is configured for.

Compatibility
There are no compatibility issues with this function, however, please note that the DCX-PCI controllers have no built-in analog inputs.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes
Delphi: function MCGetAnalog(hCtlr: HCTRLR; channel: Word): Word; stdcall;
VB: Function MCGetAnalog(ByVal hCtrlr As Integer, ByVal channel As Integer) As Integer
LabVIEW:



MCCL Reference
TA

See Also
MCSetAnalog( ), **MCPARAMEX** structure definition

MCGetDigitalIO

MCGetDigitalIO( ) returns the current state of the specified digital I/O channel.

```
short int MCGetDigitalIO(
   HCTRLR hCtlr,                  // controller handle
   WORD channel                   // channel number
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
channel Digital channel number to get state of.

Returns
The return value is TRUE if the channel is "on." A return value of FALSE indicates the channel is "off".

Comments
This function will read the current state of both input and output digital I/O channels. Note that this function simply reports if the channel is "on" or "off"; depending upon how a channel has been configured "on" (and conversely "off") may represent either a high or a low voltage level.

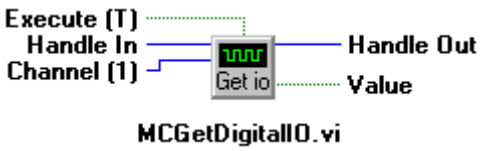
The field DigitalIO in the **MCPARAMEX** structure contains the total number of digital I/O channels the controller is configured for.

Under the MCAPI, the DC2-STN controller's input channels are numbered 1 - 8, and the output channels are numbered 9 - 16 (the MCAPI requires that each channel have a unique channel number).

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes
Delphi: function MCGetDigitalIO(hCtlr: HCTRLR; channel: Word): SmallInt; stdcall;
VB: Function MCGetDigitalIO(ByVal hCtrlr As Integer, ByVal channel As Integer) As Integer
LabVIEW:



MCCL Reference
TC

See Also
MCEnableDigitalIO( ), **MCGetDigitalIO( )**, **MCGetDigitalIOConfig( )**

MCGetDigitalIOConfig

MCGetDigitalIOConfig( ) returns the current configuration (in / out / high / low) of the specified digital I/O channel.

```
short int MCGetDigitalIO(
   HCTRLR hCtlr,                  // controller handle
   WORD channel,                  // channel number
   WORD* pMode                    // variable to hold the channel settings
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
channel Digital channel number to get configuration of.
pMode Pointer to a variable to hold the current configuration settings of the specified channel. This variable will contain one or more of the following flags on return:

Value	Description
MC_DIO_INPUT	Channel configured for input.
MC_DIO_OUTPUT	Channel configured for output.

Value	Description
MC_DIO_LOW	Channel configured for low true logic level.
MC_DIO_HIGH	Channel configured for high true logic level.
MC_DIO_LATCH	Input channel configured for latched operation.
MC_DIO_FIXED	Channel is a fixed input or output and cannot be changed using **MCConfigureDigitalIO( )**.
MC_DIO_LATCHABLE	Input channel is capable of latched operation.
MC_DIO_STEPPER	Input channel has been dedicated to driving a stepper motor (DC2-PC or DC2-STN).

Returns
The current configuration of the specified digital I/O channel is placed in the variable specified by the pointer *pMode*, and MCERR_NOERROR is returned if there were no errors. If there was an error, one of the MCERR_xxxx error codes is returned, and the variable pointed to by *pMode* is left unchanged.

Comments
The configuration of the specified channel is returned as one or more of the MC_DIO_xxx constants OR'ed together. This value is identical to the value you would create to configure the channel using **MCConfigureDigitalIO( )**, with the exception of the MC_DIO_FIXED, MC_DIO_LATCHABLE, and MC_DIO_STEPPER which are read-only (i.e. **MCGetDigitalIOConfig( )** only) parameters.

Currently none of the motion controllers supported by the MCAPI allow you to read back the configuration of the digital I/O. To implement **MCGetDigitalIOConfig( )** the MCAPI "remembers" any changes made to the digital I/O using **MCConfigureDigitalIO( )**. When the MCAPI DLL is loaded into memory (at application run time), it assumes the default state power-on state for all the installed digital I/O. Therefore, this function is most useful within a single application, after you have explicitly configured each I/O channel.

The field **DigitalIO** in the **MCPARAMEX** structure contains the total number of digital I/O channels the controller is configured for.

Under the MCAPI, the DC2-STN controller's input channels are numbered 1 - 8, and the output channels are numbered 9 - 16 (the MCAPI requires that each channel have a unique channel number).

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: mcapi32.lib
Version: MCAPI 2.1 or higher

Prototypes
Delphi: function MCGetDigitalIOConfig(hCtlr: HCTRLR; channel: Word; var pMode: Word): LongInt; stdcall;
VB: Function MCGetDigitalIOConfig(ByVal hCtrlr As Integer, ByVal channel As Integer, mode As Integer) As Long
LabVIEW: Not Supported

MCCL Reference
None

See Also
MCConfigureDigitalIO( ), **MCEnableDigitalIO( )**, **MCPARAMEX** structure definition

MCSetAnalog

MCGetAnalog( ) reads the current input state of the specified input Channel.

```
void MCSetAnalog(
   HCTRLR hCtlr,                  // controller handle
   WORD channel,                  // channel number
   WORD value                     // new output value
);
```

Parameters
hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
channel	Analog output channel number to set
value	New output value.

Returns
This function does not return a value.

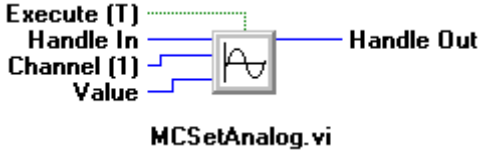
Comments
Analog output ports on MC500 and MC520 Analog Modules accept values in the range of 0 to 4095 counts (12 bits). This range of values corresponds to an output voltage of 0 to 5V or -10 to +10V, depending upon how the output is configured (see your controller's hardware manual). Each digital bit corresponds to a voltage level as follows:

Output Used	Volts per Count
0 to 5V	0.0012V
-10 to +10V	0.0049V

Compatibility
Analog output channels are not supported by the DC2-PC100 dedicated 2 axis controllers.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes
Delphi: procedure MCSetAnalog(hCtlr: HCTRLR; channel, value: Word); stdcall;
VB: Sub MCSetAnalog(ByVal hCtrlr As Integer, ByVal channel As Integer, ByVal value As Integer)
LabVIEW:



MCCL Reference
OA

See Also
MCGetAnalog( )

MCWaitForDigitalIO

MCWaitForDigitalIO( ) waits for the specified digital I/O channel to go on or off, depending upon the value of *state*.

```
void MCWaitForDigitalIO(
   HCTRLR hCtlr,                  // controller handle
   WORD channel,                  // digital I/O channel to watch
   short int state                // state of channel to watch for
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
channel Digital channel number to wait for.
state Selects state of channel to wait for:

Value	Description
TRUE	Wait for channel to go "on."
FALSE	Wait for channel to go "off."

Returns
This function does not return a value.

Comments
Digital channels 1 to 16 are built into each controller. Additional digital channels, beginning with channel 17, may be added in blocks of 16 channels using MC400 Digital I/O Modules. The field **DigitalIO** in the **MCPARAMEX** structure contains the total number of digital channels installed on the controller.

Once this command is issued, the calling program will not be able to communicate with the board until the digital I/O is equal to *state*. We recommend creating your own looping structure based on **MCGetDigitalIO( )** instead.

Compatibility
There are no compatibility issues with this function.

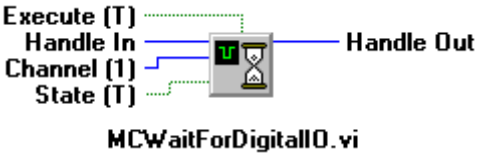
Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure MCWaitForDigitalIO(hCtlr: HCTRLR; channel: Word; state: SmallInt); stdcall;

VB: Sub MCWaitForDigitalIO(ByVal hCtrlr As Integer, ByVal channel As Integer, ByVal state As Integer)

LabVIEW:



MCCL Reference

WF, WN

See Also

MCConfigureDigitalIO( ), **MCEnableDigitalIO( )**, **MCGetDigitalIO( )**, **MCPARAMEX** structure definition

Chapter Contents

Chapter 16

Macros and Multi-tasking Functions

Macro and multi-tasking functions provide access to the motion controllers on-board macro capability, as well as the multitasking features of advanced controllers.

To see examples of how the functions in this chapter are used, please refer to the online Motion Control API Reference.

MCCancelTask

MCCancelTask( ) cancels an executing task on a multi-tasking controller. The task should have been previously started with an **MCBlockBegin( )** / **MCBlockEnd( )** pair.

```
long int MCCancelTask(
   HCTRLR hCtlr,                 // controller handle
   long int taskID               // ID of task to cancel
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
taskID	Task ID value for the task to be stopped. This value was returned by the **MCBlockEnd( )** function when the task was generated.

Returns

This function returns MCERR_NOERROR if there were no errors. One of the MCERR_xxxx defined error codes will be returned if there was a problem.

Comments

MCCancelTask( ) is the only way to stop tasks that are not programmed to stop themselves (i.e. infinite loop tasks).

See the description of **MCBlockBegin( )** for more information and reference the online help for examples.

Compatibility

The DC2 and DCX-PC100 controllers do not support background tasks.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes
Delphi: function MCCancelTask(hCtlr: HCTRLR; taskID: Longint): Longint; stdcall;
VB: Function MCCancelTask(ByVal hCtrlr As Integer, ByVal taskID As Long) As Long
LabVIEW: Not Supported

MCCL Reference
ET

See Also
MCBlockBegin( ), **MCCancelTask( )**

MCMacroCall

MCMacroCall( ) causes a previously loaded macro to be executed.

```
void MCMacroCall(
   HCTRLR hCtlr,                  // controller handle
   WORD macro                     // macro number
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
macro Macro number to execute.

Returns
This function does not return a value.

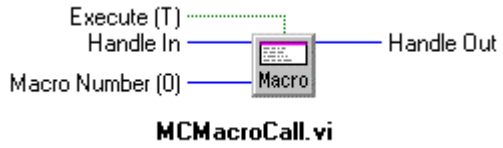
Comments
Macros are normally downloaded using the **pmcputs( )** ASCII interface command, using the Motion Control Command Language (MCCL); or by converting the MCAPI functions to a macro with the **MCBlockBegin( )** / **MCBlockEnd( )** functions. These controller level macros are often the only efficient way to implement hardware specific sequences, such as special homing routines, initializing encoder positions, etc.

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure MCMacroCall(hCtlr: HCTRLR; macro: Word); stdcall;
VB: Sub MCMacroCall(ByVal hCtrlr As Integer, ByVal macro As Integer)
LabVIEW:



MCCL Reference

MC

See Also

MCBlockBegin( ), **MCBlockEnd( )**, **pmcputs( )**, Controller hardware manual

MCRepeat

MCRepeat( ) inserts a repeat command into a block command - task, compound command, or macro.

```
long int MCRepeat(
   HCTRLR hCtlr,                  // controller handle
   long int count                 // repeat count
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
count	Repeat count. Commands that precede the **MCRepeat( )** in the block command will be repeated *count* more times (for a total execution of *count* + 1).

Returns

MCRepeat( ) returns the value MCERR_NOERROR if the function completed without errors. If there was an error, one of the MCERR_xxxx error codes is returned.

Comments

This function may only be used within an **MCBlockBegin( )** / **MCBlockEnd( )** command pair.

Compatibility

There are no compatibility issues with this function.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes

Delphi: function MCRepeat(hCtlr: HCTRLR; count: Longint): Longint; stdcall;
VB: Function MCRepeat(ByVal hCtrlr As Integer, ByVal count As Long) As Long

LabVIEW: Not Supported

MCCL Reference

RP

See Also

MCBlockBegin( ), **MCBlockEnd( )**

Chapter Contents

Chapter 17

MCAPI Driver Functions

Driver functions handle driver related housekeeping, and as such do not directly affect the controller.

To see examples of how the functions in this chapter are used, please refer to the online Motion Control API Reference.

MCBlockBegin

MCBlockBegin( ) initiates a block command sequence. All commands up to the subsequent **MCBlockEnd( )** will be included in the block. Block commands include compound commands, macro definition commands, contour path motions, and tasks on multitasking controllers.

```
long int MCBlockBegin(
   HCTRLR hCtlr,                 // controller handle
   long int mode,                // block mode type
   long int num                  // macro / task number / controlling axis
);
```

Parameters

hCtlr Controller handle, returned by a successful call to **MCOpen( )**.

mode Type of block command to begin:

Value	Description
MC_BLOCK_COMPOUND	Specifies that this block is a compound command.
MC_BLOCK_TASK	Specifies this block as an individual task on multitasking controllers. *num* should be set to the desired task number.
MC_BLOCK_MACRO	Specifies this block as a macro definition. *num* should be set to the desired macro number for this macro.
MC_BLOCK_RESETM	Resets macro memory. Setting *num* to zero resets all macros (and works with all controllers), *num* may also be set to 1 or 2 on the DCX AT200 to selectively delete ram or flash based macros.

Value	Description
MC_BLOCK_CANCEL	Cancels a block command without sending any commands to the controller.
MC_BLOCK_CONTR_USER	Specifies that this block is a user defined contour path motion. *num* should be set to the controlling axis number.
MC_BLOCK_CONTR_LIN	Specifies that this block is a linear contour path motion. *num* should be set to the controlling axis number.
MC_BLOCK_CONTR_CW	Specifies that this block is a clockwise arc contour path motion. *num* should be set to the controlling axis number.
MC_BLOCK_CONTR_CCW	Specifies that this block is a counter clockwise arc contour path motion. *num* should be set to the controlling axis number.

num Specifies the macro number for macro blocks, the task number for task blocks, the controlling axis for contour blocks, or the macro types for macro reset.

Returns

This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments

The **MCBlockBegin( )** and **MCBlockEnd( )** commands are used to bracket other API commands in order to affect how those commands are executed. While the high level MCAPI is function based (as are most Windows APIs), PMC's motion control cards are command based. They are capable of accepting single commands or blocks of commands, depending upon the complexity of the motion. To provide the same block functionality to the MCAPI the **MCBlockBegin( )** and **MCBlockEnd( )** functions where created. These functions may be used to bracket one or more MCAPI function calls to create function blocks.

One use is to create a compound command block - where multiple commands are sent to the controller as a single block. This is useful for data capture sequences, homing sequences, or anywhere you want to synchronize a complex group of commands.

For multi-tasking controllers, the block commands can be used to group individual commands as separate tasks. Multi-tasking permits multiple user programs to run in parallel on PMC's advanced motion control cards. Multi-tasking also permits you to run command sequences that would normally lock-up the controller's command interpreter in the background, thus leaving the command interpreter unaffected.

A third use of the block commands is to store the bracketed command sequence as a macro. Macros may be replayed at any time using the **MCMacroCall( )** function. Please note that API commands that read data from a controller, such as any of the **MCGet…** functions, should not be included in macros. Macro memory may be reset (cleared) by calling **MCBlockBegin( )** with Mode set to MC_BLOCK_RESETM. If your controller allows you to reset selected blocks of macros you may specify this by setting *num* to 1 for RAM-based macros or 2 for Flash memory macros.

All calls to **MCBlockBegin( )**, except those with a *mode* of MC_BLOCK_RESETM or MC_BLOCK_CANCEL require a corresponding call to **MCBlockEnd( )**. Calls to **MCBlockBegin( )** may not be nested, except that **MCBlockBegin( )** calls with an Mode of MC_BLOCK_CANCEL may be included within other **MCBlockBegin( )** blocks (this call terminates the outer **MCBlockBegin( )**, so no **MCBlockEnd( )** is needed in this case).

Beginning with version 2.0 of the MCAPI, blocks are also used for multi-axis contouring. Contouring requires first that the selected axes be placed in contouring mode and a controlling axis specified. This is done with the **MCSetOperatingMode( )** function. Then blocks of contour path moves are issued. Under the MCAPI, these contour path blocks are specified by bracketing **MCArcCenter( )**, **MCGoHome( )**, **MCMoveAbsolute( )**, **MCMoveRelative( )**, or **MCSetVectorVelocity( )** with block commands that are one of the MC_BLOCK_CONTR_xxx types.

Block commands may be canceled prior to issuing an **MCBlockEnd( )** by calling **MCBlockBegin( )** with Mode set to MC_BLOCK_CANCEL.

Compatibility
The MCAPI does not does not support contouring on the DC2, DCX-PC100, or DCX-PCI100 controllers. The DC2 and DCX-PC100 controllers do not support background tasks.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes
Delphi: function MCBlockBegin(hCtlr: HCTRLR; mode, num: Longint): Longint; stdcall;
VB: Function MCBlockBegin(ByVal hCtrlr As Integer, ByVal mode As Long, ByVal num As Long) As Long
LabVIEW: Not Supported

MCCL Reference
CP, GT, MD, RM

See Also
MCBlockEnd( ), **MCCancelTask( )**, **MCMacroCall( )**, **MCRepeat( )**

MCBlockEnd

MCBlockEnd( ) ends a block command and transmits the compound command, task, macro, or contour path to the controller.

```
long int MCBlockEnd(
   HCTRLR hCtlr,                // controller handle
   long int* pTaskID            // task ID for MC_BLOCK_TASK blocks
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
pTaskID Pointer to variable to hold the Task ID value for MC_BLOCK_TASK blocks, this parameter is ignored and may be set to NULL for MC_BLOCK_COMPOUND or MC_BLOCK_MACRO blocks. Setting this parameter to NULL for MC_BLOCK_TASK will cause the function to not return the Task ID for this task.

Returns
This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments
The **MCBlockBegin( )** and **MCBlockEnd( )** commands are used to bracket other API commands in order to affect how those commands are executed.

See the description of **MCBlockBegin( )** for more information.

Compatibility
The MCAPI does not does not support contouring on the DC2, DCX-PC100, or DCX-PCI100 controllers. The DC2 and DCX-PC100 controllers do not support background tasks.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes
Delphi: function MCBlockEnd(hCtlr: HCTRLR; var pTaskID: LongInt): Longint; stdcall;
VB: Function MCBlockEnd(ByVal hCtrlr As Integer, taskID As Long) As Long
LabVIEW: Not Supported

MCCL Reference
None

See Also
MCBlockBegin( ), **MCCancelTask( )**

MCClose

MCClose( ) closes the specified motion controller handle, and is typically called at the end of a program.

```
short int MCClose(
   HCTRLR hCtlr                  // controller handle
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.

Returns
This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

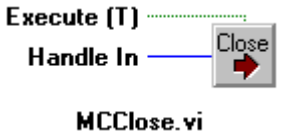
Comments
Following a call to **MCClose( )**, no further calls should be made to the Motion Control API functions with this handle (the exception being **MCOpen( )**, which may be called to open or reopen the API at any time).

By calling **MCClose( )** you notify Windows that you are done with the controller and device driver. When the last user has closed the driver Windows is then free to unload the driver from memory. Failure to call close leaves the handle open, reducing the number of available controller handles for other applications.

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes
Delphi: function MCClose(hCtlr: HCTRLR): SmallInt; stdcall;
VB: Function MCClose(ByVal hCtrlr As Integer) As Integer
LabVIEW:



MCCL Reference
None

See Also
MCOpen( )

MCGetConfigurationEx

MCGetConfigurationEx( ) obtains the configuration for the specified controller. Configuration information includes the controller type, number and type of installed motor modules, and if the controller supports scaling, contouring, etc.

```
long int MCGetConfigurationEx(
   HCTRLR hCtlr,                  // controller handle
   MCPARAMEX* pParam              // address of extended configuration
                                  // structure
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
pParam Points to an **MCPARAMEX** structure that receives the configuration information for *hCtlr*.

Returns
This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments
This function allows the application to query the driver about installed controller hardware and capabilities. Included are the number and type of axes, digital and analog IO channels, scaling, and contouring.

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: mcapi32.lib
Version: MCAPI 3.0 or higher

Prototypes
Delphi: function MCGetConfigurationEx(hCtlr: HCTRLR; var pParam: MCPARAMEX): LongInt; stdcall;
VB: Function MCGetConfigurationEx(ByVal hCtrlr As Integer, param As MCParamEx) As Long
LabVIEW: Not Supported

MCCL Reference
Dual Port RAM

See Also
MCPARAMEX structure definition

MCGetVersion

MCGetVersion( ) returns version information about the MCAPI.DLL and, optionally, about the device driver in use for a particular controller.

```
DWORD MCGetVersion(
   HCTRLR hCtlr                  // controller handle
);
```

Parameters
hCtlr Controller handle, selects which motion controller to obtain device driver version info from. May be NULL (if NULL **MCGetVersion( )** version number info is returned for the MCAPI DLL only).

Returns
The return version number for the MCAPI DLL and, if *hCtlr* is not NULL, the version number for the device driver in use for the controller. If *hCtlr* is NULL, device driver version info will be zero.

Comments
The DLL version number is contained in the low order word of the return value. The major version number is stored as the low order byte of this word, while the release number is multiplied by 10, added to the revision number, and stored as the high order byte.

If the controller handle is not NULL, the version information for the device driver that is associated with this controller will be placed in the high order word of the return value, using the same format as was used for the DLL version information.

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: mcapi32.lib
Version: MCAPI 1.2 or higher

Prototypes
Delphi: function MCGetVersion(hCtlr: HCTRLR): Longint; stdcall;
VB: Function MCGetVersion(ByVal hCtrlr As Integer) As Long
LabVIEW: Not Supported

MCCL Reference
None

MCOpen

MCOpen( ) returns a handle to a particular controller for use with subsequent API calls.

```
HCTRLR MCOpen(
   short int id,                  // controller ID
   WORD mode,                     // open mode - ASCII / binary
   char* pName                    // not used
);
```

Parameters
id Controller ID, selects the controller to open.
mode I/O mode to open controller in:

Value	Description
MC_OPEN_ASCII	Open controller for ASCII (character) I/O.
MC_OPEN_BINARY	Open the binary command interface of the controller.
MC_OPEN_EXCLUSIVE	May be OR'ed with MC_OPEN_ASCII or MC_OPEN_BINARY to request exclusive access to the controller.

pName Should be set to NULL for the present

Returns
This function returns handle to the specified controller for use in subsequent API calls. The handle will be greater than zero if the open call succeeds or less than zero if there is an error. Standard error codes (see the file MCERR.H) will be multiplied by -1 to make their values negative and returned in place of a handle, if there is an error:

Value	Description
MCERR_ALLOC_MEM	Unable to allocate memory for handle.
MCERR_CONSTANT	The constant value supplied for *mode* is invalid.
MCERR_INIT_DRIVER	Unable to initialize device driver.

MCERR_MODE_UNAVAIL	The requested mode (ASCII or binary) is unavailable. Typically due to the fact that another process has an open handle to the controller in the opposite mode.
MCERR_NO_CONTROLLER	No controller is installed at this ID, run MCSETUP.
MCERR_NOT_PRESENT	The specified controller hardware is missing or not responding.
MCERR_OPEN_EXCLUSIVE	Unable to open controller for exclusive use - another process must already have an open handle to this controller.
MCERR_OUT_OF_HANDLES	The driver is out of handles, try closing unused handles first.
MCERR_RANGE	Specified *id* is out of range.
MCERR_UNSUPPORTED_MODE	The requested open mode (ASCII or binary) is not supported for this controller.

Please note that the error codes in the table above, when an error has occurred, will returned as a negative value.

Comments

Always save the handle returned by **MCOpen( )** and use that value in subsequent calls to the API. **MCOpen( )** must be called before any other API calls are attempted. If a call is made to any other API function with a bad handle, a handle error message (MCERR_CONTROLLER) will be broadcast to all windows. Everyone is notified in the case of a bad handle because the MCAPI normally uses the handle to route error messages, and obviously can't do this if the handle is invalid.

If it is necessary that no one else gains access to a controller while you are using it, you may combine the open mode with MC_OPEN_EXCLUSIVE:

```
if ((hCtlr = MCOpen( 7, MC_OPEN_ASCII | MC_OPEN_EXCLUSIVE, NULL )) > 0)
{
   // got an exclusive handle
}
```

will only return a valid handle if no other process has an open handle to this controller already, and will prevent any one else from opening the controller while the exclusive handle is open.

The name argument in the **MCOpen( )** function call is for future enhancements to the API and should be set to NULL for the present.

If you are using an DCX-AT or DCX-PCI configured for multi-interface, you may open binary and ASCII handles simultaneously. Exclusive handles are interface based, not controller based, in this case (i.e. you may have one exclusive ASCII handle and one exclusive binary handle open at the same time).

Compatibility

There are no compatibility issues with this function.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: function MCOpen(id: SmallInt; mode: Word; pName: PChar): HCTRLR; stdcall;
VB: Function MCOpen(ByVal id As Integer, ByVal mode As Integer, ByVal name As String) As Integer

LabVIEW:

MCCL Reference
None

See Also
MCClose( ), **MCErrorNotify( )**

MCReopen

MCReopen( ) may be used to change the mode of an existing handle.

```
long int MCReopen(
   HCTRLR hCtlr,                  // controller handle
   WORD mode                      // new mode
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
mode New mode flags:

Value	Description
MC_OPEN_ASCII	Open controller for ASCII (character) I/O.
MC_OPEN_BINARY	Open the binary command interface of the controller.
MC_OPEN_EXCLUSIVE	May be combined with MC_OPEN_ASCII or MC_OPEN_BINARY using the binary or operator '|' to request exclusive access.

Returns
MCReopen( ) returns the value MCERR_NOERROR, if the function completed without errors. If there was an error, one of the MCERR_xxxx error codes is returned.

Comments
The most likely cause for failure is that another open handle exists for the same controller. **MCReopen( )** cannot change a controller's open mode if there are multiple handles, as there is no way to notify the owners of those other handles that a mode switch has occurred. If you plan on using this function in an application, it is suggested that you open the controller in exclusive mode to prevent any additional handles from being opened.

If you are using a DCX-PCI or DCX-AT in multi-interface mode, the above restrictions do not apply.

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes
Delphi: function MCReopen(hCtlr: HCTRLR; mode: Word): Longint; stdcall;
VB: Function MCReopen(ByVal hCtrlr As Integer, ByVal mode As Integer) As Long
LabVIEW: Not Supported

MCCL Reference
None

See Also
MCClose( ), **MCOpen( )**

MCSetTimeoutEx

MCSetTimeoutEx( ) sets the timeout period for I/O to a particular controller.

```
long int MCSetTimeoutEx(
   HCTRLR hCtlr,                // controller handle
   double timeout,              // new timeout value
   double* pOldTimeout          // old timeout value
);
```

Parameters
hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
timeout New timeout period, in seconds.
pOldTimeout Pointer to a double precision floating point variable that will hold the old timeout setting for the specified axis. If the pointer is NULL, no value is returned.

Returns
If there were no errors, the previous timeout setting is placed in the variable specified by the pointer *pOldTimeout*, and MCERR_NOERROR is returned. If there was an error, one of the MCERR_xxxx error codes is returned, and the variable pointed to by *pOldTimeout* is left unchanged. If the pointer *pOldTimeout* is NULL, the old timeout value is not returned.

Comments
The timeout period is the maximum amount of time, in seconds, that the MCAPI device driver will wait to send a command and/or receive a reply. The default setting for timeout for all controllers is zero seconds. A timeout setting of zero will cause the controller to wait forever (i.e. no timeout) for I/O to complete.

Note that a timeout value that is acceptable for most functions may fail (i.e. timeout) if the controller is asked to perform a lengthy operation (a long wait, a reset, etc.). One option in these cases is to change the timeout value for the duration of the long operation, then change the timeout value back.

Compatibility
There are no compatibility issues with this function.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: mcapi32.lib
Version: MCAPI 1.3 or higher

Prototypes

Delphi: function MCSetTimeoutEx(hCtlr: HCTRLR; timeout: Double; var pOldTimeout: Double): Longint; stdcall;
VB: Function MCSetTimeoutEx(ByVal hCtrlr As Integer, ByVal timeout As Double, oldTimeout As Double) As Long
LabVIEW: Not Supported

MCCL Reference

None

Chapter Contents

Chapter

18

MCAPI OEM Low Level Functions

The OEM low level commands provide direct access to controller functionality. The functions in this group are not part of the formal Motion Control API.

These functions have been implemented in a way that is consistent with DOS mode libraries for these controllers. This consistency is designed to simplify the task of porting existing DOS applications to Windows.

To see examples of how the functions in this chapter are used, please refer to the online Motion Control API Reference.

pmccmd

pmccmd( ) downloads a formatted binary command buffer directly to the PMC controller. Programmers should use the more advanced **pmccmdex( )** instead of this function when possible.

```
long int pmccmd(
   HCTRLR hCtlr,                // controller handle
   short int bytes,             // length of buffer
   void* pBuffer                // pointer to command buffer
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
bytes	Length of buffer, in bytes.
pBuffer	Pointer to command buffer.

Returns

The return value from this function is the actual number of bytes downloaded. Because of the nature of the binary interface, the return value will be equal to the buffer size (value of the *bytes* argument), indicating the command buffer was successfully downloaded, or zero, indicating a problem communicating with the controller.

Comments
The binary interface is described in detail in the hardware manual that accompanied your controller. The user of this function is responsible for correctly formatting the buffer - no checking is performed by the function. To send binary commands to the motion controller the *hCtlr* handle must have opened in binary mode.

This function may be used within an **MCBlockBegin( )** / **MCBlockEnd( )** pair to create Macros, Compound commands, or Tasks.

This command function may also be used in ASCII mode; in this case the command buffer should contain a correctly formatted ASCII command (including the terminating carriage return "\r").

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h and mccl.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes
Delphi: function pmccmd(hCtlr: HCTRLR; bytes: SmallInt; pBuffer: PChar): SmallInt; stdcall;
VB: Function pmccmd(ByVal hCtrlr As Integer, ByVal bytes As Integer, ByVal buffer As String) As Integer
LabVIEW: Not Supported

MCCL Reference
None

See Also
pmcrdy( ), **pmcrpy( )**

pmccmdex

pmccmdex( ) downloads a formatted binary command buffer directly to the PMC controller.

```
long int pmccmdex(
   HCTRLR hCtlr,                  // controller handle
   WORD axis,                     // Axis number for this command
   WORD cmd,                      // MCCL command
   void* pArgument,               // pointer to command argument
   long int type                  // type of argument
);
```

Parameters
hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
axis	Axis number for this command.
cmd	MCCL command to execute - see MCCL.H and the User's Manual for your motion controller.
pArgument	Pointer to a variable that has the argument for this command.

type Type of data pointed to by *pArgument*:

Value	Description
MC_TYPE_LONG	Indicates *pArgument* points to a variable of type long integer.
MC_TYPE_DOUBLE	Indicates *pArgument* points to a variable of type double precision floating point.
MC_TYPE_FLOAT	Indicates *pArgument* points to a variable of type single precision floating point.
MC_TYPE_REG	Indicates *pArgument* points to a variable of the format of a 32 bit integer with register number.
MC_TYPE_NONE	Indicates *pArgument* points to a variable of type which is NULL.

Returns
This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments
The binary interface is described in detail in the hardware manual that accompanied your controller. To send binary commands to the motion controller the *hCtlr* handle must have opened in binary mode.

This function may be used within an **MCBlockBegin( )** / **MCBlockEnd( )** pair to create Macros, Compound commands, or Tasks.

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcapi.h and mccl.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 2.2 or higher

Prototypes
Delphi: function pmccmdex(hCtlr: HCTRLR; axis: Word; cmd: Word; var pArgument: Pointer; type: Longint): Longint; stdcall;
VB: Function pmccmdex(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal cmd As Integer, argument As Any, ByVal argtype As Long) As Long
LabVIEW: Not Supported

MCCL Reference
None

See Also
pmcrdy( ), **pmcrpyex( )**

pmcgetc

pmcgetc( ) reads a single character from the controller ASCII interface.

```
short int pmcgetc(
   HCTRLR hCtlr                  // controller handle
);
```

Parameters

hCtlr Controller handle, returned by a successful call to **MCOpen( )**.

Returns

The return value from this function is number of bytes actually read from the controller (1 or 0).

Comments

This function will return immediately if there is no character available. Use the string get command, **pmcgets( )**, if you want to wait for a character, or place **pmcgetc( )** in a loop.

You must open the controller in ASCII mode (MC_OPEN_ASCII) in order to use this command.

Compatibility

There are no compatibility issues with this function.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: function pmcgetc(hCtlr: HCTRLR): SmallInt; stdcall;
VB: Function pmcgetc(ByVal hCtrlr As Integer) As Integer
LabVIEW: Not Supported

MCCL Reference

None

See Also

pmcgetc( ), **pmcputc( )**, **pmcputs( )**

pmcgetram

pmcgetram( ) reads *bytes* from controller memory beginning at location *offset*.

```
short int pmcgetram(
   HCTRLR hCtlr,                  // controller handle
   WORD offset,                   // memory offset to read from
   void* pBuffer,                 // buffer to hold ram value
   short int bytes                // number of bytes of memory to read
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
offset	Starting memory location, relative to the beginning of controller dual ported ram, to read from.
pBuffer	Buffer to hold read in controller memory, must be at least *bytes* long.
bytes	Number of bytes of memory to read.

Returns

This function does not return a value.

Comments

No range checking is performed on *offset* or *bytes* - it is the caller's responsibility to supply valid values for these arguments. Consult the controller hardware manual for details on the controller memory map.

Do not use this command within an **MCBlockBegin( )** / **MCBlockEnd( )** block.

Compatibility

There are no compatibility issues with this function.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure pmcgetram(hCtlr: HCTRLR; offset: Word; pBuffer: PChar; bytes: SmallInt); stdcall;
VB: Sub pmcgetram(ByVal hCtrlr As Integer, ByVal offset As Integer, ByVal buffer As String, ByVal bytes As Integer)
LabVIEW: Not Supported

MCCL Reference

None

See Also

pmcputram( )

pmcgets

pmcgets( ) reads a null-terminated ASCII string of up to *bytes* characters from the controller ASCII interface.

```
short int pmcgets(
   HCTRLR hCtlr,                 // controller handle
   void* pBuffer,                // pointer to buffer
   short int bytes               // length of buffer
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
pBuffer	Pointer to reply buffer.
bytes	Length of *buffer*, in bytes.

Returns

The return value from this function is number of bytes actually read from the controller.

Comments

This function will wait for a reply for as long as the controller is busy processing command. A zero will be returned when the controller is idle and there are no reply characters. However, a non-zero timeout value will force the function to return the number of characters it has received prior to the timeout.

You must open the controller in ASCII mode (MC_OPEN_ASCII) in order to use this command.

Compatibility

There are no compatibility issues with this function.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: function pmcgets(hCtlr: HCTRLR; pBuffer: PChar; bytes: SmallInt): SmallInt; stdcall;
VB: Function pmcgets(ByVal hCtrlr As Integer, ByVal buffer As String, ByVal bytes As Integer) As Integer
LabVIEW: Not Supported

MCCL Reference

None

See Also

MCSetTimeoutEx( ), **pmcgetc( )**, **pmcputc( )**, **pmcputs( )**

pmcputc

pmcputc( ) writes a single character to the controller ASCII interface.

```
short int pmcputc(
   HCTRLR hCtlr,                  // controller handle
   short int char                 // output char
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
char	Character to output.

Returns

This function returns a one if the character is successfully written or a zero if it is unable to write to the controller.

Comments

Remember to terminate all command strings with a carriage return "\r" in order for the command to be executed. This command does not wait for the controller - if it is unable to write the character it returns immediately with a return value of zero.

You must open the controller in ASCII mode (MC_OPEN_ASCII) in order to use this command.

Do not use this command within an **MCBlockBegin( )** / **MCBlockEnd( )** block. This function attempts to write immediately to the motion controller.

Compatibility

There are no compatibility issues with this function.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: function pmcputc(hCtlr: HCTRLR; char: SmallInt): SmallInt; stdcall;
VB: Function pmcputc(ByVal hCtrlr As Integer, ByVal char As Integer) As Integer
LabVIEW: Not Supported

MCCL Reference

None

See Also

pmcgetc( ), **pmcgets( )**, **pmcputs( )**

pmcputram

pmcputram( ) writes *bytes* directly into the controller's memory beginning at location *offset*.

```
void pmcputram(
   HCTRLR hCtlr,                  // controller handle
   WORD offset,                   // memory offset to write to
   void* pBuffer,                 // buffer to hold ram value
   short int bytes                // number of bytes of memory to write
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
offset	Starting memory location, relative to the beginning of controller dual ported ram, to write to.
pBuffer	Buffer of data to write into controller memory.
bytes	Number of bytes of memory to write.

Returns

This function does not return a value.

Comments

No range checking is performed on *offset* or *bytes*. It is the caller's responsibility to supply valid values for these arguments. Writing directly to dual ported ram can cause unpredictable results. **USE THIS FUNCTION WITH EXTREME CAUTION!**

Compatibility

There are no compatibility issues with this function.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: procedure pmcputram(hCtlr: HCTRLR; offset: Word; pBuffer: PChar; bytes: SmallInt); stdcall;
VB: Sub pmcputram(ByVal hCtrlr As Integer, ByVal offset As Integer, ByVal buffer As String, ByVal bytes As Integer)
LabVIEW: Not Supported

MCCL Reference

None

See Also

pmcgetram( )

pmcputs

pmcputs( ) writes a NULL terminated command string to the controller ASCII interface.

```
short int pmcputs(
   HCTRLR hCtlr,                  // controller handle
   char* pBuffer                  // output string
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
pBuffer	Output string.

Returns

This function returns the number of characters actually written to the controller. This number may be less than the length of the string if the controller becomes busy and stops accepting characters.

Comments

Remember to terminate all command strings with a carriage return "\r" in order for the command to be executed. This function consumes any reply characters from the controller while it is writing (this may change in future implementations).

You must open the controller in ASCII mode (MC_OPEN_ASCII) in order to use this command.

Compatibility

There are no compatibility issues with this function.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: function pmcputs(hCtlr: HCTRLR; pBuffer: PChar): SmallInt; stdcall;
VB: Function pmcputs(ByVal hCtrlr As Integer, ByVal buffer As String) As Integer
LabVIEW: Not Supported

MCCL Reference

None

See Also

pmcgetc( ), **pmcgets( )**, **pmcputs( )**

pmcrdy

pmcrdy( ) checks the specified controller to see if it is ready to accept a binary command buffer.

```
short int pmcrdy(
   HCTRLR hCtlr                  // controller handle
);
```

Parameters

hCtlr Controller handle, returned by a successful call to **MCOpen( )**.

Returns

The return value from this function is TRUE (+1) if the controller is ready to accept commands. The controller will return FALSE if it is busy. For the AT200 controller, a value of -1 is returned if the controller is ready to accept data in file download mode.

Comments

Basic language users are cautioned that Visual Basic defines TRUE as -1, while Windows defines TRUE to be +1 (the API uses the Windows value for TRUE and returns a +1 if the controller is ready). Therefore, code such as:

```
   if pmcrdy( hCtlr ) = True then
```

will not work as expected in Visual Basic.

Compatibility

There are no compatibility issues with this function.

Requirements

Header: include mcapi.h, mcapi.pas, or mcapi32.bas
Library: mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi: function pmcrdy(hCtlr: HCTRLR): SmallInt; stdcall;
VB: Function pmcrdy(ByVal hCtrlr As Integer) As Integer
LabVIEW: Not Supported

MCCL Reference

None

See Also

pmccmd( ), **pmcrpy( )**

pmcrpy

pmcrpy( ) reads a binary reply of up to *bytes* bytes from the controller. Programmers should use the more advanced **pmcrpyex( )** instead of this function when possible.

```
long int pmcrpy(
   HCTRLR hCtlr,                  // controller handle
   short int bytes,               // length of buffer
   void* pBuffer                  // pointer to buffer
);
```

Parameters

hCtlr	Controller handle, returned by a successful call to **MCOpen( )**.
bytes	Length of buffer, in bytes.
pBuffer	Pointer to reply buffer.

Returns

The return value from this function is the actual number of bytes read. This value may be less than the argument *bytes*, but will never exceed *bytes*. If the controller has no reply ready, the return value will be zero.

Comments

This function waits for a reply for as long as the controller is busy - it returns with a return value of zero if no reply is (or will be) available.

You must open the controller in ASCII mode (MC_OPEN_ASCII) in order to use this command.

Compatibility

There are no compatibility issues with this function.

Requirements

Header: include mcapi.h and mccl.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 1.0 or higher

Prototypes

Delphi:	function pmcrpy(hCtlr: HCTRLR; bytes: SmallInt; pBuffer: PChar): SmallInt; stdcall;
VB:	Function pmcrpy(ByVal hCtrlr As Integer, ByVal bytes As Integer, ByVal buffer As String) As Integer
LabVIEW:	Not Supported

MCCL Reference

None

See Also

pmccmd( ), **pmcrdy( )**, **pmcrpyex( )**

pmcrpyex

pmcrpyex( ) reads a binary reply of up to *bytes* bytes from the controller.

```
long int pmcrpyex(
   HCTRLR hCtlr,                  // controller handle
   void* pReply,                  // pointer to command reply
   long int type                  // type of argument
);
```

Parameters

hCtlr Controller handle, returned by a successful call to **MCOpen( )**.
pReply Pointer to a variable to hold the reply value.
type Type of data pointed to by *pReply*:

Value	Description
MC_TYPE_LONG	Indicates *pReply* points to a variable of type long integer.
MC_TYPE_DOUBLE	Indicates *pReply* points to a variable of type double precision floating point.

Returns

This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was a problem.

Comments

The binary interface is described in detail in the hardware manual that accompanied your controller.

Compatibility

There are no compatibility issues with this function.

Requirements

Header: include mcapi.h and mccl.h, mcapi.pas, or mcapi32.bas
Library: use mcapi32.lib
Version: MCAPI 2.2 or higher

Prototypes

Delphi: function pmcrpyex(hCtlr: HCTRLR; var pReply: Pointer; type: Longint): Longint; stdcall;
VB: Function pmcrpyex(ByVal hCtrlr As Integer, reply As Any, ByVal argtype As Long) As Long
LabVIEW: Not Supported

MCCL Reference

None

See Also

pmccmdex( ), **pmcrdy( )**, **pmcrpy( )**

Chapter Contents

Chapter

19

MCAPI Common Motion Dialog Functions

The Common Motion Dialog library includes easy-to-use high-level functions for the control and configuration of your motion controller. By combining these functions in a single library we've made it easy for programmers to include the Common Motion Dialog functionality in their application programs. Functions are provided for the configuration of servo and stepper axes, scaling setup, controller selection, file download, and save/restore of motor settings.

To see examples of how the functions in this chapter are used, please refer to the online Motion Control API Reference.

MCDLG_AboutBox

MCDLG_AboutBox( ) displays a simple About dialog box that includes version information about both the application and the Motion Control API.

```
long int MCDLG_AboutBox(
   HWND hWnd,                    // handle to parent window
   LPCSTR title,                 // title string for the dialog box
   long int bitmapID             // bitmap ID for the dialog box
);
```

Parameters

hWnd	Handle to parent window of About Box. This handle is used by **MCDLG_AboutBox( )** to retrieve VERSIONINFO strings from the application.
title	An optional title string for the About dialog box. If this pointer is NULL or points to a zero length string the default title of “About” is used.
bitmapID	An optional Bitmap resource identifier. If greater than zero, the specified bitmap will be displayed in the About dialog box. If zero, **MCDLG_AboutBox( )** will display the default bitmap. Bitmaps should be no larger than 240 (width) by 80 (height) pixels, 16 colors.

Returns

This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was an error creating the dialog box.

Comments

Version information is obtained by retrieving VERSIONINFO values from the executable module. The specific strings queried for are “CompanyName”, “FileDescription”, “FileVersion”, and “LegalCopyright”. It is a good idea to include a VERSIONINFO resource in any application as it permits Windows to accurately determine the version of any executable file or DLL. Applications and DLLs supplied with the Motion Control API include a VERSIONINFO resource.

The dialog box displays a default logo bitmap above the version information. By specifying a valid bitmap resource ID for the *bitmapID* parameter you may change the bitmap displayed. If this parameter is greater than zero the new bitmap will replace the default in the About dialog box. Bitmaps should be no larger than 240 (width) by 80 (height) pixels, 16 colors.

If a NULL pointer or a pointer to a zero length string is passed as the *title* argument the default title will be used. Acceptance of a pointer to a zero length string was included to support programming languages that have difficulty with NULL pointers (e.g. Visual Basic). To eliminate the title pass a pointer to a string with a single space (i.e. " ").

Note that **MCDLG_AboutBox( )** uses the HWND argument passed to it to identify the executable file from which to read the VERSIONINFO information. In some development environments, such as Visual Basic, window handles are owned by a DLL supplied by the author of the development system, not the user's EXE file. In these situations, **MCDLG_AboutBox( )** is unable to correctly perform its VERSIONINFO query and should not be used.

Compatibility

There are no compatibility issues with this function.

Requirements

Header: include mcdlg.h, mccdlg.pas, or mcdlg32.bas
Library: use mcdlg32.lib and mcapi32.lib
Version: MCAPI 2.1 or higher

Prototypes

Delphi: function MCDLG_AboutBox(hWnd: HWnd; title: PChar; bitmapID: Longint): Longint; stdcall;
VB: Function MCDLG_AboutBox(ByVal hWnd As Long, ByVal title As String, ByVal bitmapID As Long) As Long
LabVIEW: Not Supported

MCDLG_CommandFileExt

MCDLG_CommandFileExt( ) returns the file extension for MCCL command files for a particular motion controller type.

```
long int MCDLG_CommandFileExt(
   long int type,                // controller type identifier
   long int flags,               // flags
   LPCSTR buffer,                // buffer for file extension string
   long int length               // length of string buffer, in bytes
);
```

Parameters

type Motion Controller type, must be equal to one of the predefined motion controller types (see MCAPI.H).
flags Reserved for future use (set to zero).

buffer	Pointer to a string buffer that will hold the file extension (should be _MAX_FILE long).
length	Size of buffer, in bytes.

Returns

This function returns a pointer to the file extension string for the specified motion controller type. It returns NULL if *type* does not specify a valid controller type.

Comments

The Motion Control API registers a separate file extension for each controller type. The MCAPI tools, such as Win Control, use these file extensions when they open MCCL command files. You can use this function to get the registered file extension for any controller type.

See the MCAPI sample program Win Control for an example.

Compatibility

There are no compatibility issues with this function.

Requirements

Header: include mcdlg.h, mccdlg.pas, or mcdlg32.bas
Library: use mcdlg32.lib and mcapi32.lib
Version: MCAPI 3.0 or higher

Prototypes

Delphi:	function MCDLG_CommandFileExt(type: LongInt; flags: LongInt; buffer: PChar; length: Longint): PChar; stdcall;
VB:	Function MCDLG_CommandFileExt(ByVal argtype As Long, ByVal flags As Long, ByVal buffer As String, ByVal length As Long) As String
LabVIEW:	Not Supported

MCDLG_ConfigureAxis

MCDLG_ConfigureAxis( ) displays a servo or stepper axis setup dialog that permits user configuration of the axis.

```
long int MCDLG_ConfigureAxis(
   HWND hWnd,                    // handle to parent window
   HCTRLR hCtlr,                 // handle to a motion controller
   WORD axis,                    // axis number to configure
   long int flags,               // configuration flags
   LPCSTR title                  // optional axis title for the dialog box
);
```

Parameters

hWnd	Handle to parent window. May be NULL.
hCtlr	Motion Controller handle, returned by a successful call to MCOpen( ).
axis	Axis number of axis to be configured.
flags	Flags to control the operation (multiple flags may be OR'ed together):

Value	Description
MCDLG_CHECKACTIVE	Checks if an axis is moving before the new settings are written to the controller and skips if the axis is moving. Combine with MCDLG_PROMPT to prompt user whether or not to proceed.
MCDLG_PROMPT	Combine with MCDLG_CHECKACTIVE to prompt user whether or not to proceed if a motor is moving and the user has dismissed the dialog box with OK.

title An optional title string for the axis. If this pointer is NULL or points to a zero length string the default title, which includes the axis number and a description of the axis type is used.

Returns

This function returns MCERR_NOERROR if the user pressed OK button to dismiss the dialog box. It returns MCERR_CANCEL if the user pressed the CANCEL button to dismiss the dialog box. It returns one of the other MCERR_xxxx error codes if there was an error creating the dialog box.

Comments

This function provides comprehensive, ready-to-use setup dialogs for stepper and servo motor axis types. The motion controller is queried for the current axis settings to initialize this dialog box. Any changes the user makes are sent to the motion controller if the user dismisses the dialog by pressing the OK button.

Changing the parameters of an axis while it is moving may result in erratic behavior (such as when you choose to include the motor position in the changed parameters). The flag MCDLG_CHECKACTIVE forces this function to check the axis to see if it is active before it proceeds. By default MCDLG_CHECKACTIVE will skip the changing of an active axis, but if you also include the flag MCDLG_PROMPT the user will be prompted for how to proceed. The programming samples are all built with MCDLG_CHECKACTIVE and MCDLG_PROMPT set.

If a NULL pointer or a pointer to a zero length string is passed as the *title* argument, the default title will be used. Acceptance of a pointer to a zero length string was included to support programming languages that have difficulty with NULL pointers (e.g. Visual Basic). To eliminate the title pass a pointer to a string with a single space (i.e. " ").

Compatibility

There are no compatibility issues with this function.

Requirements

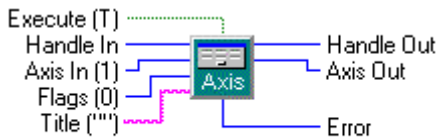
Header: include mcdlg.h, mccdlg.pas, or mcdlg32.bas
Library: use mcdlg32.lib and mcapi32.lib
Version: MCAPI 2.1 or higher

Prototypes

Delphi: function MCDLG_ConfigureAxis(hWnd: HWnd; hCtlr: HCTRLR; axis: Word; flags: Longint; title: PChar): Longint; stdcall;

VB: Function MCDLG_ConfigureAxis(ByVal hWnd As Long, ByVal hCtlr As Integer, ByVal axis As Integer, ByVal flags As Long, ByVal title As String) As Long

LabVIEW:



MCDLG_ConfigureAxis.vi

MCDLG_ControllerDescEx

MCDLG_ControllerDescEx( ) returns a descriptive string for the specified motion controller type.

```
LPCSTR MCDLG_ControllerDescEx(
   long int type,                // controller type identifier
   long int flags,               // flags
   LPSTR buffer,                 // buffer for descriptive string
   long int length               // size of buffer, in bytes
);
```

Parameters

type Motion Controller type, must be equal to one of the predefined motion controller types (see MCAPI.H).

flags Flags to control the operation:

Value	Description
MCDLG_NAMEONLY	Resulting string will contain only the name portion (no description).
MCDLG_DESCONLY	Resulting string will contain only the name portion (no name).

buffer Pointer to a string buffer that will hold the descriptive string.

length Size of *buffer*, in bytes.

Returns

This function returns a pointer to the descriptive string buffer for the specified motion controller type, or it returns NULL if *type* does not specify a valid controller type.

Comments

This extended version of **MCDLG_ControllerDesc( )** includes by default the controller name and a description of the controller in the output string. Use the *flags* parameter to control the information included in the string.

You may use this function to provide a descriptive string for a motion controller by passing the function the **ControllerType** member of an **MCPARAMEX** structure following a call to **MCGetConfigurationEx( )**. As an example, the MCDLG function **MCDLG_ControllerInfo( )** uses this function to produce its Controller Information dialog.

Compatibility

There are no compatibility issues with this function.

Requirements

Header: include mcdlg.h, mccdlg.pas, or mcdlg32.bas
Library: use mcdlg32.lib and mcapi32.lib
Version: MCAPI 3.0 or higher

Prototypes

Delphi: function MCDLG_ControllerDescEx(type: LongInt; flags: LongInt; buffer: PChar; length: Longint): PChar; stdcall;

VB: Function MCDLG_ControllerDescEx(ByVal argtype As Long, ByVal flags As Long, ByVal buffer As String, ByVal length As Long) As String
LabVIEW: Not Supported

MCDLG_ControllerInfo

MCDLG_ControllerInfo( ) displays configuration information about the specified motion controller.

```
long int MCDLG_ControllerInfo(
   HWND hWnd,                    // handle to parent window
   HCTRLR hCtlr,                 // handle to a motion controller
   long int flags,               // configuration flags
   LPCSTR title                  // title for the dialog box
);
```

Parameters

hWnd Handle to parent window. May be NULL.
hCtlr Motion Controller handle, returned by a successful call to **MCOpen( )**.
flags Currently no flags are defined for **MCDLG_ControllerInfo( )**, and this argument should be set to zero.
title An optional title string for the dialog box. If this pointer is NULL or points to a zero length string, a default title is used.

Returns

This function returns MCERR_NOERROR if there were no errors, or it returns one of the MCERR_xxxx defined error codes if there was an error creating the dialog box.

Comments

This function displays a read only dialog providing information on the current motion controller configuration and capabilities (this information is typically used by programs to control execution for example can the controller multi-task? Is contouring supported?).

If a NULL pointer or a pointer to a zero length string is passed as the *title* argument the default title will be used. Acceptance of a pointer to a zero length string was included to support programming languages that have difficulty with NULL pointers (e.g. Visual Basic). To eliminate the title pass a pointer to a string with a single space (i.e. " ").

Compatibility

There are no compatibility issues with this function.

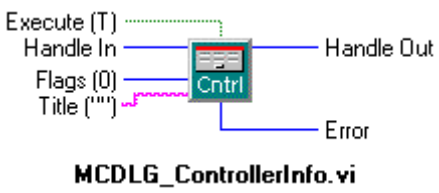
Requirements

Header: include mcdlg.h, mccdlg.pas, or mcdlg32.bas
Library: use mcdlg32.lib and mcapi32.lib
Version: MCAPI 2.1 or higher

Prototypes

Delphi: function MCDLG_ControllerInfo(hWnd: HWnd; hCtlr: HCTRLR; flags: Longint; title: PChar): Longint; stdcall;
VB: Function MCDLG_ControllerInfo(ByVal hWnd As Long, ByVal hCtlr As Integer, ByVal flags As Long, ByVal title As String) As Long

LabVIEW:



MCDLG_DownloadFile

MCDLG_DownloadFile( ) downloads an ASCII command file to the specified motion controller.

```
long int MCDLG_DownloadFile(
   HWND hWnd,                    // handle of window to echo download to
   HCTRLR hCtlr,                 // handle of motion controller
   long int flags,               // configuration flags
   LPCSTR fileName               // path/filename of file to download
);
```

Parameters

hWnd	Handle of window to echo downloaded characters to. May be NULL.
hCtlr	Motion Controller handle, returned by a successful call to **MCOpen( )**.
flags	Currently no flags are defined for **MCDLG_ConfigureAxis( )**, and this field should be left blank.
fileName	Path / filename of file to download.

Returns

This function returns MCERR_NOERROR if the file was successfully downloaded, or it returns one of the other MCERR_xxxx error codes if there was an error downloading the file.

Comments

MCDLG_DownloadFile( ) opens the specified file and downloads the contents to the specified controller. If a valid (non-NULL) window handle is given for *hWnd*, downloaded characters (and replies from the controller) are sent to the window via WM_CHAR messages. This feature allows you to use **MCDLG_DownloadFile( )** with a terminal interface application, such as Win Control, that displays the file while it is being downloaded.

Compatibility

There are no compatibility issues with this function.

Requirements

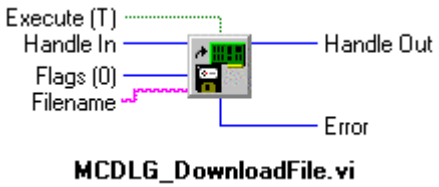
Header: include mcdlg.h, mccdlg.pas, or mcdlg32.bas
Library: use mcdlg32.lib and mcapi32.lib
Version: MCAPI 2.1 or higher

Prototypes

Delphi: function MCDLG_DownloadFile(hWnd: HWnd; hCtlr: HCTRLR; flags: Longint; fileName: PChar): Longint; stdcall;

VB: Function MCDLG_DownloadFile(ByVal hWnd As Long, ByVal hCtlr As Integer, ByVal flags As Long, ByVal fileName As String) As Long

LabVIEW:



MCDLG_Initialize

MCDLG_Initialize( ) must be called before any other MCDLG functions are called or any of the MCDLG window classes are used.

```
long int MCDLG_Initialize(
   void
);
```

Returns

This function returns MCERR_NOERROR if the MCDLG library was successfully initialized, or it returns one of the other MCERR_xxxx error codes if there was an error initializing the library.

Comments

Calling **MCDLG_Initialize( )** ensures that internal MCDLG data structures are correctly initialized and that MCDLG window classes are registered.

Compatibility

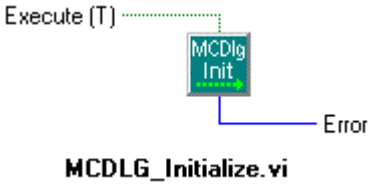
There are no compatibility issues with this function.

Requirements

Header: include mcdlg.h, mccdlg.pas, or mcdlg32.bas
Library: use mcdlg32.lib and mcapi32.lib
Version: MCAPI 2.1 or higher

Prototypes

Delphi: function MCDLG_Initialize: Longint; stdcall;
VB: Function MCDLG_Initialize( ) As Long
LabVIEW:



MCDLG_ListControllers

MCDLG_ListControllers( ) enumerates the types of motion controllers installed.

```
long int MCDLG_ListControllers(
   short int idArray[ ],         // pointer to an array for controller type
                                 // IDs
   short int size                // size of idArray[]
);
```

Parameters

idArray	Pointer to an array of short integers, filled with controller types on return.
size	Size of *idArray[ ]* (number of integers).

Returns

The return value is the number of installed controllers found.

Comments

MCDLG_ListControllers( ) fills *idArray[ ]* with controller type identifiers, where the type of the controller configured at ID 0 is stored in idArray[0], the type of the controller configured at ID 1 is stored in idArray[1], etc. In order to list all installed controllers the array must have a size of at least MC_MAX_ID + 1 (the constant MC_MAX_ID is defined in the MCAPI header files).

Compatibility

There are no compatibility issues with this function.

Requirements

Header: include mcdlg.h, mccdlg.pas, or mcdlg32.bas
Library: use mcdlg32.lib and mcapi32.lib
Version: MCAPI 2.1 or higher

Prototypes

Delphi:	function MCDLG_ListControllers(idArray: Array of SmallInt; size: SmallInt): Longint; stdcall;
VB:	Function MCDLG_ListControllers Lib "mcdlg32.dll" (idArray As Any, ByVal size As Integer) As Long
LabVIEW:	Not Supported

MCDLG_ModuleDescEx

MCDLG_ModuleDescEx( ) returns a descriptive string for the specified module/axis type.

```
LPCSTR MCDLG_ModuleDescEx(
   long int type,                // axis type identifier
   long int flags,               // flags
   LPSTR buffer,                 // buffer for descriptive string
   long int length               // size of buffer, in bytes
);
```

Parameters

type Module type, must be equal to one of the predefined module types (see MCAPI.H).

flags Flags to control the operation:

Value	Description
MCDLG_NAMEONLY	Resulting string will contain only the name portion (no description).
MCDLG_DESCONLY	Resulting string will contain only the description portion (no name).

buffer Pointer to a string buffer that will hold the descriptive string.

length Size of *buffer*, in bytes.

Returns

This function returns pointer to the descriptive string buffer for the specified axis type, or it returns NULL if *type* does not specify a valid axis type.

Comments

This extended version of **MCDLG_ModuleDesc( )** includes by default the module name and a description of the module in the output string. Use the *flags* parameter to control the information included in the string.

You may use this function to provide a descriptive string for an axis by passing the function the **ModuleType** member of an **MCAXISCONFIG** structure following a call to **MCGetAxisConfiguration( )**. As an example, the MCDLG function **MCDLG_ConfigureAxis( )** uses this function to produce its default axis description string.

Compatibility

There are no compatibility issues with this function.

Requirements

Header: include mcdlg.h, mccdlg.pas, or mcdlg32.bas
Library: use mcdlg32.lib and mcapi32.lib
Version: MCAPI 3.0 or higher

Prototypes

Delphi: function MCDLG_ModuleDescEx(type: LongInt; flags: LongInt; buffer: PChar; length: Longint): PChar; stdcall;

VB: Function MCDLG_ModuleDescEx(ByVal argtype As Long, ByVal flags As Long, ByVal buffer As String, ByVal length As Long) As String

LabVIEW: Not Supported

MCDLG_RestoreAxis

MCDLG_RestoreAxis( ) restores the settings of the given axis to a previously saved state.

```
long int MCDLG_RestoreAxis(
   HCTRLR hCtlr,                  // handle to a motion controller
   WORD axis,                     // axis number to configure
   long int flags,                // configuration flags
   LPCSTR privateIniFile          // optional INI file to read from
);
```

Parameters

hCtlr Motion Controller handle, returned by a successful call to **MCOpen( )**.

axis Axis number of axis to be restored.

flags Flags to control the restore operation (multiple flags may be OR'ed together):

Value	Description
MCDLG_CHECKACTIVE	Checks if an axis is moving before the settings are restored and skips if the axis is moving. Combine with MCDLG_PROMPT to prompt user whether or not to proceed.
MCDLG_NOMOTION	Do not restore **MCMOTIONEX** structure settings.
MCDLG_NOFILTER	Do not restore **MCFILTEREX** structure settings.
MCDLG_NOPHASE	Do not restore phase setting.
MCDLG_NOPOSITION	Do not restore axis position.
MCDLG_PROMPT	If the stored data doesn't match the type of the axis being restored to a Message Box will be displayed. Also affects the behavior of MCDLG_CHECKACTIVE (see above).

privateIniFile Name, optionally with path and drive, of the INI file in which to save the axis settings. If NULL **MCDLG_RestoreAxis( )** will use MCAPI.INI.

Returns

This function returns MCERR_NOERROR if there were no problems, or it returns one of the other MCERR_xxxx error codes if there was an error. The most common reason for a return value of FALSE is supplying an invalid or non-existent filename for *privateIniFile*.

Comments

MCDLG_SaveAxis( ) encodes the motion controller type and module type into signature that is saved with the axis settings. **MCDLG_RestoreAxis( )** checks for a valid signature before restoring the axis settings. If you make changes to your hardware configuration (i.e. change module types or controller type) **MCDLG_RestoreAxis( )** will refuse to restore those settings.

You may specify the constant MC_ALL_AXES for the *axis* parameter in order to restore the parameters for all axes installed on a motion controller with a single call to this function.

Restoring the parameters to an axis while it is moving may result in erratic behavior (such as when you choose to include the motor position in the restored parameters). The flag MCDLG_CHECKACTIVE forces this function to check each restored axis to see if it is active before it proceeds. By default MCDLG_CHECKACTIVE will skip the restore of an active axis, but if you also include the flag MCDLG_PROMPT the user will be prompted for how to proceed. The programming samples are all built with MCDLG_CHECKACTIVE and MCDLG_PROMPT set.

Note that this function writes a lot of information to the motion controller for each axis saved, and should be used sparingly over slow interfaces such as the RS232.

If a NULL pointer or a pointer to a zero length string is passed as the *privateIniFile* argument the default file (MCAPI.INI) will be used. Most applications should use the default file so that configuration data may be easily shared among

applications. Acceptance of a pointer to a zero length string was included to support programming languages that have difficulty with NULL pointers (e.g. Visual Basic).

Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcdlg.h, mccdlg.pas, or mcdlg32.bas
Library: use mcdlg32.lib and mcapi32.lib
Version: MCAPI 2.1 or higher

Prototypes
Delphi: function MCDLG_RestoreAxis(hCtlr: HCTRLR; axis: Word; flags: Longint; privateIniFile: PChar): Longint; stdcall;
VB: Function MCDLG_RestoreAxis(ByVal hCtlr As Integer, ByVal axis As Integer, ByVal flags As Long, ByVal privateIniFile As String) As Long
LabVIEW:

See Also
MCDLG_SaveAxis( )

MCDLG_RestoreDigitalIO

MCDLG_RestoreDigitalIO( ) restores the settings of the all the digital I/O channels between *startChannel* and *endChannel* (inclusive) to their previously saved states.

```
long int MCDLG_RestoreDigitalIO(
   HCTRLR hCtlr,                  // handle to a motion controller
   WORD startChannel,             // starting channel number to restore
   WORD endChannel,               // ending channel number to restore
   LPCSTR privateIniFile          // optional INI file to read from
);
```

Parameters
hCtlr Motion Controller handle, returned by a successful call to **MCOpen( )**.
startChannel Number of the first digital I/O channel axis to be restored. If set to zero the first available channel on the controller will be used.
endChannel Number of the last digital I/O channel axis to be restored. If set to zero the last available channel on the controller will be used.
privateIniFile Name, optionally with path and drive, of the INI file in which to save the axis settings. If NULL **MCDLG_RestoreDigitalIO( )** will use MCAPI.INI.

Returns
This function returns MCERR_NOERROR if the settings were restored correctly, or it returns MCERR_RANGE if either StartChannel or EndChannel is out of range.

Comments
By setting *startChannel* and *endChannel* both to zero this function will automatically restore all the digital I/O channels on a motion controller.

If a NULL pointer or a pointer to a zero length string is passed as the *privateIniFile* argument, the default file (MCAPI.INI) will be used. Most applications should use the default file so that configuration data may be easily shared among applications. Acceptance of a pointer to a zero length string was included to support programming languages that have difficulty with NULL pointers (e.g. Visual Basic).

Under the MCAPI, the DC2-STN controller's input channels are numbered 1 - 8, and the output channels are numbered 9 - 16 (the MCAPI requires that each channel have a unique channel number).

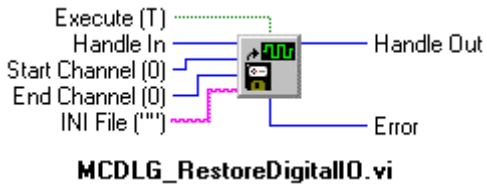
Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcdlg.h, mccdlg.pas, or mcdlg32.bas
Library: use mcdlg32.lib and mcapi32.lib
Version: MCAPI 2.1 or higher

Prototypes
Delphi: function MCDLG_RestoreDigitalIO(hCtlr: HCTRLR; startChannel: Word; endChannel: Word; privateIniFile: PChar):Longint; stdcall;

VB: Function MCDLG_RestoreDigitalIO(ByVal hCtlr As Integer, ByVal startChannel As Integer, ByVal endChannel As Integer, ByVal privateIniFile As String) As Long

LabVIEW:



See Also
MCDLG_SaveDigitalIO( )

MCDLG_SaveAxis

MCDLG_SaveAxis( ) saves the settings of the given axis to an initialization file for later use.

```
long int MCDLG_SaveAxis(
   HCTRLR hCtlr,                 // handle to a motion controller
   WORD axis,                    // axis number to configure
   long int flags,               // configuration flags
   LPCSTR privateIniFile         // optional INI file to write to
);
```

Parameters

hCtlr Motion Controller handle, returned by a successful call to **MCOpen( )**.

axis Axis number of axis to be restored.

flags Flags to control the restore operation (multiple flags may be OR'ed together):

Value	Description
MCDLG_NOMOTION	Do not restore **MCMOTIONEX** structure settings.
MCDLG_NOFILTER	Do not restore **MCFILTEREX** structure settings.
MCDLG_NOPHASE	Do not restore phase setting.
MCDLG_NOPOSITION	Do not restore axis position.

privateIniFile Name, optionally with path and drive, of the INI file in which to save the axis settings. If NULL **MCDLG_RestoreAxis( )** will use MCAPI.INI.

Returns

This function returns MCERR_NOERROR if there were no problems, or it returns one of the other MCERR_xxxx error codes if there was an error. The most common reason for a return value of FALSE is supplying an invalid or non-existent filename for *privateIniFile*.

Comments

MCDLG_SaveAxis( ) encodes the motion controller type and module type into signature that is saved with the axis settings. **MCDLG_RestoreAxis( )** checks for a valid signature before restoring the axis settings. If you make changes to your hardware configuration (i.e. change module types or controller type) **MCDLG_RestoreAxis( )** will refuse to restore those settings.

You may specify the constant MC_ALL_AXES for the *axis* parameter in order to save the parameters for all axes installed on a motion controller with a single call to this function. Setting *axis* to -1 will cause **MCDLG_SaveAxis( )** to delete all of the stored axis information for this controller.

Note that this function reads a lot of information from the motion controller for each axis saved, and should be used sparingly over slow interfaces such as the RS232.

If a NULL pointer or a pointer to a zero length string is passed as the *privateIniFile* argument the default file (MCAPI.INI) will be used. Most applications should use the default file so that configuration data may be easily shared among applications. Acceptance of a pointer to a zero length string was included to support programming languages that have difficulty with NULL pointers (e.g. Visual Basic).

Compatibility

There are no compatibility issues with this function.

Requirements

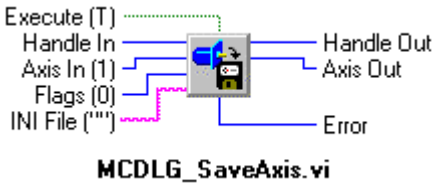
Header: include mcdlg.h, mccdlg.pas, or mcdlg32.bas
Library: use mcdlg32.lib and mcapi32.lib
Version: MCAPI 2.1 or higher

Prototypes

Delphi: function MCDLG_SaveAxis(hCtlr: HCTRLR; axis: Word; flags: Longint; privateIniFile: PChar): Longint; stdcall;

VB: Function MCDLG_SaveAxis(ByVal hCtlr As Integer, ByVal axis As Integer, ByVal flags As Long, ByVal privateIniFile As String) As Long

LabVIEW:



MCDLG_SaveDigitalIO

MCDLG_SaveDigitalIO( ) saves the settings of the all the digital I/O channels between *startChannel* and *endChannel* (inclusive) to an INI file.

```
long int MCDLG_SaveDigitalIO(
   HCTRLR hCtlr,                  // handle to a motion controller
   WORD startChannel,             // starting channel number to save
   WORD endChannel,               // ending channel number to save
   LPCSTR privateIniFile          // optional INI file to write to
);
```

Parameters

hCtlr	Motion Controller handle, returned by a successful call to **MCOpen( )**.
startChannel	Number of the first digital I/O channel axis to be restored. If set to zero the first available channel on the controller will be used.
endChannel	Number of the last digital I/O channel axis to be restored. If set to zero, the last available channel on the controller will be used.
privateIniFile	Name, optionally with path and drive, of the INI file in which to save the axis settings. If NULL **MCDLG_SaveDigitalIO( )** will use MCAPI.INI.

Returns

MCERR_NOERROR if the settings were saved correctly or MCERR_RANGE if either *startChannel* or *endChannel* is out of range.

Comments

By setting *startChannel* and *endChannel* both to zero this function will automatically save all the digital I/O channels on a motion controller.

If a NULL pointer or a pointer to a zero length string is passed as the *privateIniFile* argument the default file (MCAPI.INI) will be used. Most applications should use the default file so that configuration data may be easily shared among applications. Acceptance of a pointer to a zero length string was included to support programming languages that have difficulty with NULL pointers (e.g. Visual Basic).

Under the MCAPI, the DC2-STN controller's input channels are numbered 1 - 8, and the output channels are numbered 9 - 16 (the MCAPI requires that each channel have a unique channel number).

Compatibility

There are no compatibility issues with this function.

Requirements

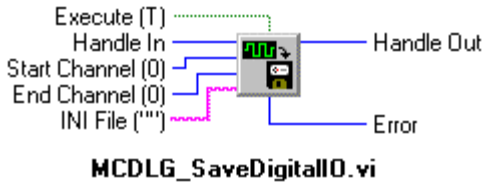
Header: include mcdlg.h, mccdlg.pas, or mcdlg32.bas
Library: use mcdlg32.lib and mcapi32.lib
Version: MCAPI 2.1 or higher

Prototypes

Delphi: function MCDLG_SaveDigitalIO(hCtlr: HCTRLR; startChannel: Word; endChannel: Word; privateIniFile: PChar):Longint; stdcall;

VB: Function MCDLG_SaveDigitalIO(ByVal hCtlr As Integer, ByVal startChannel As Integer, ByVal endChannel As Integer, ByVal privateIniFile As String) As Long

LabVIEW:



MCDLG_Scaling

MCDLG_Scaling( ) displays a scaling setup dialog and, if the motion controller supports scaling, allows the user to change the scaling parameters.

```
long int MCDLG_Scaling(
   HWND hWnd,                    // handle to parent window
   HCTRLR hCtlr,                 // handle to a motion controller
   WORD axis,                    // axis number to configure
   long int flags,               // configuration flags
   LPCSTR title                  // optional title for the dialog box
);
```

Parameters

hWnd Handle to parent window. May be NULL.
hCtlr Motion Controller handle, returned by a successful call to **MCOpen( )**.
axis Axis number of axis to be scaled.
flags Flags to control scaling:

Value	Description

MCDLG_PROMPT	If user clicks OK to dismiss dialog display a message warning that scaling changes will take effect following the next motor on command.

title An optional title string for the About dialog box. If this pointer is NULL or points to a zero length string the default title of “About” is used.

Returns
This function returns MCERR_NOERROR if the user pressed OK button to dismiss the dialog box. It returns MCERR_CANCEL if the user pressed the CANCEL button to dismiss the dialog box, or it returns one of the other MCERR_xxxx error codes if there was an error creating the dialog box.

Comments
For controllers that don't support scaling the Motion Control API will fill in the **MCSCALE** data structure with default values (zero for offsets, one for factors). **MCDLG_Scaling( )** will display these defaults as read-only. For advanced controllers such as the DCX-AT and the DCX-PCI **MCDLG_Scaling( )** will display the current scale factors and allow the user to change them.

If a NULL pointer or a pointer to a zero length string is passed as the *title* argument the default title will be used. Acceptance of a pointer to a zero length string was included to support programming languages that have difficulty with NULL pointers (e.g. Visual Basic). To eliminate the title pass a pointer to a string with a single space (i.e. " ").

NOTE: Scaling changes will take effect following the next motor on command (**MCEnableAxis( )**) after **MCDLG_Scaling( )** completes.

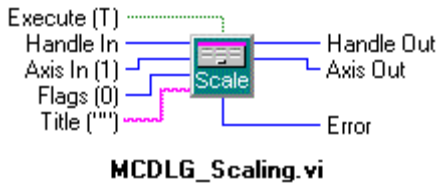
Compatibility
There are no compatibility issues with this function.

Requirements
Header: include mcdlg.h, mccdlg.pas, or mcdlg32.bas
Library: use mcdlg32.lib and mcapi32.lib
Version: MCAPI 2.1 or higher

Prototypes
Delphi: function MCDLG_Scaling(hWnd: HWnd; hCtlr: HCTRLR; axis: Word; flags: Longint; title: PChar): Longint; stdcall;

VB: Function MCDLG_Scaling(ByVal hWnd As Long, ByVal hCtlr As Integer, ByVal axis As Integer, ByVal flags As Long, ByVal title As String) As Long

LabVIEW:



MCDLG_Scaling.vi

MCDLG_SelectController

MCDLG_SelectController( ) displays a list of installed controllers and allows the user to select a controller from the list.

```
long int MCDLG_SelectController(
   HWND hWnd,                    // handle to parent window
   short int currentID,          // ID of currently selected controller
   long int flags,               // configuration flags
   LPCSTR title                  // optional title for the dialog box
);
```

Parameters

hWnd	Handle to parent window. May be NULL.
currentID	ID of the motion controller currently in use. In the selection list, this controller will be highlighted. Set to -1 to ignore.
flags	Currently no flags are defined for **MCDLG_ConfigureAxis( )**, and this field should be left blank.
title	An optional title string for the dialog box. If this pointer is NULL or points to a zero length string the default title is used.

Returns

This function returns a controller ID if the user selected a controller and pressed the OK button to dismiss the dialog, or it returns a -1 if the user pressed the CANCEL button to dismiss the dialog. A value of -1 is also returned if there are no motion controllers currently configured.

Comments

This function displays a list of installed controllers and allows the user to select one from the list. If a valid ID is given for *currentID* that controller will be highlighted in the list as the default selection (set *currentID* to -1 prevent a default selection). If no motion controllers have been configured for use with the Motion Control Applet in the Motion Control Panel, a message is displayed indicating that no controllers are configured and -1 is returned to the calling program.

If a NULL pointer or a pointer to a zero length string is passed as the *title* argument the default title will be used. Acceptance of a pointer to a zero length string was included to support programming languages that have difficulty with NULL pointers (e.g. Visual Basic). To eliminate the title pass a pointer to a string with a single space (i.e. " ").

Compatibility

There are no compatibility issues with this function.

Requirements

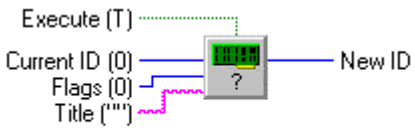
Header: include mcdlg.h, mccdlg.pas, or mcdlg32.bas
Library: use mcdlg32.lib and mcapi32.lib
Version: MCAPI 2.1 or higher

Prototypes

Delphi: function MCDLG_SelectController(hWnd: HWnd; currentID: SmallInt; flags: Longint; title: PChar): SmallInt; stdcall;

VB: Function MCDLG_SelectController(ByVal hWnd As Long, ByVal currentID As Integer, ByVal flags As Long, ByVal title As String) As Integer

LabVIEW:



MCDLG_SelectController.vi

Chapter Contents

- MCAPI Error codes

Chapter

20

MCAPI Controller Error Codes

The MCAPI defined error messages are listed numerically in the following table. Where possible corrective action is included in the description column. Please note that many MCAPI function descriptions also include information regarding errors that are specific to that function.

Error	Constant	Description
0	MCERR_NOERROR	No error has occurred.
1	MCERR_NO_CONTROLLER	No controller assigned at this ID. Use MCSETUP to configure a controller.
2	MCERR_OUT_OF_HANDLES	MCAPI driver out of handles. The driver is limited to 32 open handles. Applications that do not call **MCClose( )** when they exit may leave handles unavailable, forcing a reboot.
3	MCERR_OPEN_EXCLUSIVE	Cannot open - another application has the controller opened for exclusive use.
4	MCERR_MODE_UNAVAIL	Controller already open in different mode. Some controller types can only be open in one mode (ASCII or binary) at a time.
5	MCERR_UNSUPPORTED_MODE	Controller doesn't support this mode for **MCOpen( )** - i.e. ASCII or binary.
6	MCERR_INIT_DRIVER	Couldn't initialize the device driver.
7	MCERR_NOT_PRESENT	Controller hardware not present.
8	MCERR_ALLOC_MEM	Memory allocation error. This is an internal memory allocation problem with the DLL, contact Technical Support for assistance.
9	MCERR_WINDOWSERROR	A windows function returned an error - use **GetLastError( )** under WIN32 for details
10	-	reserved
11	MCERR_NOTSUPPORTED	Controller doesn't support this feature.
12	MCERR_OBSOLETE	Function is obsolete.
13	MCERR_CONTROLLER	Invalid controller handle.
14	MCERR_WINDOW	Invalid window handle.
15	MCERR_AXIS_NUMBER	Axis number out of range.
16	MCERR_AXIS_TYPE	Axis type doesn't support this feature.
17	MCERR_ALL_AXES	Cannot use MC_ALL_AXES for this function.
18	MCERR_RANGE	Parameter was out of range.
19	MCERR_CONSTANT	Constant value inappropriate.
20	MCERR_UNKNOWN_REPLY	Unexpected or unknown reply.
21	MCERR_NO_REPLY	Controller failed to reply.
22	MCERR_REPLY_SIZE	Reply size incorrect.
23	MCERR_REPLY_AXIS	Wrong axis for reply.
24	MCERR_REPLY_COMMAND	Reply is for different command.
25	MCERR_TIMEOUT	Controller failed to respond.
26	MCERR_BLOCK_MODE	Block mode error. Caused by calling **MCBlockEnd( )** without first calling **MCBlockBegin( )** to begin the block.
27	MCERR_COMM_PORT	Communications port (RS232) driver reported an error.
28	MCERR_CANCEL	User canceled action (such as when an MCDLG dialog box is dismissed with the CANCEL button.
29	MCERR_NOT_INITIALIZED	Feature was not correctly initialized before being enable or used.

Chapter Contents

Chapter

21

MCAPI Constants

The symbolic constants described in this section provide a safe, descriptive way of accessing the MCAPI features. The actual numeric value of these constants may change in future versions of the API, however the constant names will remain fixed. Use of these symbolic values will help to insure that future changes to the API won't break existing code. The constant values also help to produce more readable code. To find the actual value of any given constant, please refer to the online Motion Control API Reference or the MCAPI.H header file.

Constant	Description
DC2PC100	Value for the **ControllerType** member of an **MCPARAMEX** structure, it indicates that a DC2 PC100 controller is installed.
DC2SERVO	Identifies an axis as one of the dedicated servo axes on a DC2PC100 controller.
DC2STEPPER	Identifies an axis as one of the optional stepper axes on a DC2PC100 controller.
DC2STN	Value for the **ControllerType** member of an **MCPARAMEX** structure, it indicates that a DC2 STN controller is installed.
DCXPC100	Value for the **ControllerType** member of an **MCPARAMEX** structure, it indicates that a DCX series PC100 controller is installed.
DCXAT100	Value for the **ControllerType** member of an **MCPARAMEX** structure, it indicates that a DCX series AT100 controller is installed.
DCXAT200	Value for the **ControllerType** member of an **MCPARAMEX** structure, it indicates that a DCX series AT200 controller is installed.
DCXAT300	Value for the **ControllerType** member of an **MCPARAMEX** structure, it indicates that a DCX series AT300 controller is installed.
DCXPCI100	Value for the **ControllerType** member of an **MCPARAMEX** structure, it indicates that a DCX series PC100 controller is installed.
DCXPCI300	Value for the **ControllerType** member of an **MCPARAMEX** structure, it indicates that a DCX series PCI300 controller is installed.
MC_ABSOLUTE	Specifies that a position is in absolute units.
MC_ALL_AXES	When used in place of an axis number this constant implies that the command be performed on all installed axes. This option is not generally permitted on get type commands, i.e. to get the current position for all installed axes you should issue an individual **MCGetPositionEx( )** call for each axis.
MC_BLOCK_CANCEL	Argument to **MCBlockBegin( )** function canceling any commands queued (but not yet executed) as a result of a previous call to **MCBlockBegin( )**.
MC_BLOCK_COMPOUND	Argument to **MCBlockBegin( )** function specifying this block as a compound command block. Commands will not be executed until the **MCBlockEnd( )** command is issued.
MC_BLOCK_CONTR_CCW	Argument to **MCBlockBegin( )** function specifying this block as a contour path counter-clockwise arc (valid only for controllers that support contouring).
MC_BLOCK_CONTR_CW	Argument to **MCBlockBegin( )** function specifying this block as a contour path clockwise arc (valid only for controllers that support contouring).
MC_BLOCK_CONTR_LIN	Argument to **MCBlockBegin( )** function specifying this block as a contour path linear motion (valid only for controllers that support contouring).
MC_BLOCK_CONTR_USER	Argument to **MCBlockBegin( )** function specifying this block as a contour path user defined motion (valid only for controllers that support contouring).
MC_BLOCK_MACRO	Argument to **MCBlockBegin( )** function specifying this block as a macro command. All commands up to the **MCBlockEnd( )** will be included in the macro.
MC_BLOCK_RESETM	Argument to **MCBlockBegin( )** function that will cause macro storage to be cleared.

Constant	Description
MC_BLOCK_TASK	Argument to **MCBlockBegin( )** function specifying this block as separate task (valid only for controllers that support multi-tasking).
MC_CAPTURE_ACTUAL	Used to select the actual position data from the data capture functions.
MC_CAPTURE_ADVANCED	capture flag for CaptureModes member of **MCAXISCONFIG**
MC_CAPTURE_ERROR	Used to select the following error data from the data capture functions.
MC_CAPTURE_OPTIMAL	Used to select the optimal position data from the data capture functions.
MC_CAPTURE_TORQUE	Used to select the torque data from the data capture functions.
MC_COMPARE_DISABLE	Disable position compare mode, also used to disable compare output on position match.
MC_COMPARE_ENABLE	Enable position compare mode.
MC_COMPARE_STATIC	Set compare output on position match.
MC_COMPARE_TOGGLE	Toggle compare output on position match.
MC_COMPARE_INVERT	Set compare output on position match.
MC_COMPARE_ONESHOT	Set compare output on position match.
MC_COUNT_CAPTURE	Return the current captured position count.
MC_COUNT_COMPARE	Return the current compare position count.
MC_COUNT_CONTOUR	Return the current contour position count.
MC_COUNT_FILTER	Return the current digital filter coefficient count.
MC_COUNT_FILTERMAX	Return the maximum digital filter size supported.
MC_CURRENT_FULL	Restores a stepper motor current to full power. Commonly used to restore full power, prior to driving, following a reduced current setting while a stepper motor was idle. This constant is used to set the value of the **Current** member of a **MCMOTIONEX** structure.
MC_CURRENT_HALF	Reduces stepper motor current to half power. Commonly used to reduce heating when a stepper motor is not driving. This constant is used to set the value of the **Current** member of a **MCMOTIONEX** structure.
MC_DATA_ACTUAL	see MC_CAPTURE_ACTUAL.
MC_DATA_ERROR	see MC_CAPTURE_ERROR.
MC_DATA_OPTIMAL	see MC_CAPTURE_OPTIMAL.
MC_DIO_FIXED	Indicates that a digital I/O channel's I/O state (i.e. input or output) is fixed, and may not be changed with **MCConfigureDigitalIO( )**.
MC_DIO_HIGH	Configures a digital I/O channel for high true logic level when used as an argument to **MCConfigureDigitalIO( )**.
MC_DIO_INPUT	Configures a digital I/O channel for input when used as an argument to **MCConfigureDigitalIO( )**.
MC_DIO_LATCH	Configures a digital input channel for input latching when used as an argument to **MCConfigureDigitalIO( )**.
MC_DIO_LATCHABLE	Indicates that a digital I/O channel may be configured for latched input using **MCConfigureDigitalIO( )**.
MC_DIO_LOW	Configures a digital I/O channel for low true logic level when used as an argument to **MCConfigureDigitalIO( )**.
MC_DIO_OUTPUT	Configures a digital I/O channel for output when used as an argument to **MCConfigureDigitalIO( )**.
MC_DIO_STEPPER	Indicates that a digital I/O channel is configured for driving a stepper motor on a DC2-PC or DC2-STN controller

Constant	Description
MC_DIR_NEGATIVE	When operating in velocity mode this constant may be used as argument to **MCDirection( )** to select the negative travel direction. The physical relationship of MC_DIR_NEGATIVE to the actual direction of travel (or rotation) will depend upon your mechanical setup.
MC_DIR_POSITIVE	When operating in velocity mode this constant may be used as argument to **MCDirection( )** to select the positive travel direction. The physical relationship of MC_DIR_POSITIVE to the actual direction of travel (or rotation) will depend upon your mechanical setup.
MC_IM_CLOSEDLOOP	Selects the normal (open loop) input mode for MC360 Stepper Modules.
MC_IM_OPENLOOP	Selects the closed-loop input mode for MC360 Stepper Modules.
MC_INT_FREEZE	Selects the wait until move complete mode for the integral term option.
MC_INT_NORMAL	Selects the normal (always active) mode for the integral term option.
MC_INT_ZERO	Selects the zero and wait until move complete mode for the integral term option.
MC_LIMIT_ABRUPT	Selects abrupt stop mode when a limit is tripped.
MC_LIMIT_BOTH	Enables both the positive and negative limits.
MC_LIMIT_INVERT	Inverts limit logic mode for hard limits.
MC_LIMIT_MINUS	Enables the negative limit for hard and soft limits.
MC_LIMIT_OFF	Selects axis off mode when a limit is tripped.
MC_LIMIT_PLUS	Enables the positive limit for hard and soft limits.
MC_LIMIT_SMOOTH	Selects smooth stop mode when a limit is tripped.
MC_LRN_POSITION	When used as an argument to the **MCLearnPoint( )** function, this mode will cause the actual position of the axis to be stored in point memory.
MC_LRN_TARGET	When used as an argument to the **MCLearnPoint( )** function, this mode will cause the current target position of the axis to be stored in point memory.
MC_MAX_ID	Specifies the maximum allowable value for the ID parameter to the **MCOpen( )** call, where 0 <= ID <= MC_MAX_ID.
MC_MODE_CONTOUR	Selects the contouring mode of operation for an axis when used as an argument to **MCSetOperatingMode( )**.
MC_MODE_GAIN	Selects the gain mode of operation for an axis when used as an argument to **MCSetOperatingMode( )**.
MC_MODE_POSITION	Selects the position mode of operation for an axis when used as an argument to **MCSetOperatingMode( )**.
MC_MODE_TORQUE	Selects the torque mode of operation for an axis when used as an argument to **MCSetOperatingMode( )**.
MC_MODE_UNKNOWN	Return value from **MCGetOperatingMode( )** when it is unable to determine the current operating mode.
MC_MODE_VELOCITY	Selects the velocity mode of operation for an axis when used as an argument to **MCSetOperatingMode( )**.
MC_OM_BIPOLAR	Selects the bipolar output mode for MC200 Advanced Servo Modules.
MC_OM_CW_CCW	Selects the clockwise - counterclockwise output mode for MC260 Advanced Stepper Modules.
MC_OM_PULSE_DIR	Selects the pulse and direction output mode for MC260 Advanced Stepper Modules.
MC_OM_UNIPOLAR	Selects the unipolar output mode for MC200 Advanced Servo Modules.

Constant	Description
MC_OPEN_ASCII	When used as an argument to the **MCOpen( )** function it specifies that a controller is to be open for ASCII (character) based communication.
MC_OPEN_BINARY	When used as an argument to the **MCOpen( )** function it specifies that a controller is to be open for binary communication.
MC_OPEN_EXCLUSIVE	This constant may be combined with either MC_OPEN_ASCII or MC_OPEN_BINARY for calls to **MCOpen( )** to prevent other applications from gaining access to the controller while it is open with an exclusive handle.
MC_PHASE_REV	Selects reverse phasing for the servo module output when used as an argument to **MCSetServoOutputPhase( )**.
MC_PHASE_STD	Selects standard phasing for the servo module output when used as an argument to **MCSetServoOutputPhase( )**.
MC_PROF_PARABOLIC	This constant may be used as the value of the mode argument to the **MCSetProfile( )** API function. It selects the parabolic profile for acceleration and deceleration.
MC_PROF_SCURVE	This constant may be used as the value of the mode argument to the **MCSetProfile( )** API function. It selects the S-Curve profile for acceleration and deceleration.
MC_PROF_TRAPEZOID	This constant may be used as the value of the mode argument to the **MCSetProfile( )** API function. It selects the trapezoidal profile for acceleration and deceleration.
MC_PROF_UNKNOWN	This constant is returned by the **MCGetProfile( )** API function if it is unable to determine the present profile setting. The most likely cause is older firmware, contact PMC for information on firmware updates.
MC_RATE_HIGH	This constant is used as an argument to the **UpdateRate** member of an **MCFILTEREX** structure. For servo motors and closed-loop steppers, setting **UpdateRate** to this value sets the maximum feedback loop update rate. When used for an open-loop stepper motor, it sets the maximum pulse rate range. Please refer to your User Manual for product specific information.
MC_RATE_LOW	This constant is used as an argument to the **UpdateRate** member of an **MCFILTEREX** structure. For servo motors and closed-loop steppers, setting **UpdateRate** to this value sets the low feedback loop update rate. When used for an open-loop stepper motor, it sets the low pulse rate range. Please refer to your User Manual for product specific information.
MC_RATE_MEDIUM	This constant is used as an argument to the **UpdateRate** member of an **MCFILTEREX** structure. For servo motors and closed-loop steppers, setting **UpdateRate** to this value sets the middle feedback loop update rate. When used for an open-loop stepper motor, it sets the middle pulse rate range. Please refer to your User Manual for product specific information.
MC_RATE_UNKNOWN	Returned if MCAPI cannot determine the current rate.
MC_RELATIVE	Specifies that a position supplied is relative to the current axis position.
MC_STAT_ACCEL	Selects the Accelerating status bit (DC2 PC100 only).
MC_STAT_AMP_ENABLE	Selects the Amp Fault Enabled status bit (DCX controllers only).
MC_STAT_AMP_FAULT	Selects the Amp Fault status bit (DCX controllers only).
MC_STAT_AT_TARGET	Selects the At Target status bit (DC2 PC100 controllers only).
MC_STAT_BREAKPOINT	Selects the Breakpoint status bit.

Constant	Description
MC_STAT_BUSY	Selects the Busy status bit (DCX controllers only). When set indicates that dual port memory is being refreshed.
MC_STAT_CAPTURE	Selects the Position Capture status bit (DC2 PC100 controllers only).
MC_STAT_DIR	Selects the Direction status bit.
MC_STAT_EDGE_FOUND	Selects the Edge Found status bit (DCX PCI controllers only).
MC_STAT_ERROR	Selects the Motor Error status bit.
MC_STAT_FOLLOWING	Selects the Following Error status bit (DCX controllers only).
MC_STAT_FULL_STEP	Selects the Full Step status bit (DCX controllers only).
MC_STAT_HALF_STEP	Selects the Half Step status bit (DCX controllers only).
MC_STAT_HOMED	Selects the Motor Homed status bit.
MC_STAT_INDEX_FOUND	Selects the Index Found status bit (DCX PCI controllers only).
MC_STAT_INP_AMP	Selects the Amp Fault Input status bit (DCX controllers only).
MC_STAT_INP_AUX	Selects the Auxiliary Encoder Index Input status bit (DCX AT200, DCX AT300, DCX PCI controllers only).
MC_STAT_INP_HOME	Selects the Home Input status bit (DCX controllers only).
MC_STAT_INP_INDEX	Selects the Index Input status bit (DCX controllers only).
MC_STAT_INP_MJOG	Selects the Minus Jog Input status bit (DCX PC100 / DCX AT100 controllers only).
MC_STAT_INP_MLIM	Selects the Minus Limit Input status bit (DCX controllers only).
MC_STAT_INP_PJOG	Selects the Plus Jog Input status bit (DCX PC100 / DCX AT100 controllers only).
MC_STAT_INP_PLIM	Selects the Plus Limit Input status bit (DCX controllers only).
MC_STAT_INP_USER1	Selects the User #1 Input status bit (DCX AT200, DCX AT300 controllers only).
MC_STAT_INP_USER2	Selects the User #2 Input status bit (DCX AT200, DCX AT300 controllers only).
MC_STAT_JOG_ENAB	Selects the Jogging Enabled status bit (DCX AT200, DCX AT300 controllers only).
MC_STAT_JOGGING	Selects the Motor Jogging status bit (DCX PC100 / DCX AT100 controllers only).
MC_STAT_LMT_ABORT	Selects the Abort Limit Mode status bit (DC2 PC100 controllers only).
MC_STAT_LMT_STOP	Selects the Stop Limit Mode status bit (DC2 PC100 controllers only).
MC_STAT_LOOK_EDGE	Selects the Looking for Edge status bit.
MC_STAT_LOOK_INDEX	Selects the Looking for Index status bit.
MC_STAT_MJOG_ENAB	Selects the Minus Jog Enable status bit (DCX PC100 / DCX AT100 controllers only).
MC_STAT_MJOG_ON	Selects the Minus Jog On status bit (DCX PC100 / DCX AT100 controllers only).
MC_STAT_MLIM_ENAB	Selects the Minus Hard Limit Enable status bit.
MC_STAT_MLIM_TRIP	Selects the Minus Hard Limit Tripped status bit.
MC_STAT_MODE_ARC	Selects the Arc Mode status bit (DC2 PC100 controllers only).
MC_STAT_MODE_CNTR	Selects the Contouring Mode status bit (DC2 PC100 controllers only).
MC_STAT_MODE_LIN	Selects the Linear Mode status bit (DC2 PC100 controllers only).
MC_STAT_MODE_POS	Selects the Position Mode status bit (DC2 PC100 controllers only).

Constant	Description
MC_STAT_MODE_SLAVE	Selects the Slave Mode status bit (DC2 PC100 controllers only).
MC_STAT_MODE_TRQE	Selects the Torque Mode status bit (DC2 PC100 controllers only).
MC_STAT_MODE_VEL	Selects the Velocity Mode status bit.
MC_STAT_MSOFT_ENAB	Selects the Minus Soft Limit Enable status bit (DCX AT200, DCX AT300, DCX PCI controllers only).
MC_STAT_MSOFT_TRIP	Selects the Minus Soft Limit Tripped status bit (DCX AT200, DCX AT300, DCX PCI controllers only).
MC_STAT_MTR_ENABLE	Selects the Motor On status bit.
MC_STAT_NULL	Selects the NULL Stepper Position status bit (DCX PCI300 controllers only).
MC_STAT_PHASE	Selects the Phase Reversed status bit.
MC_STAT_PJOG_ENAB	Selects the Plus Jog Enable status bit (DCX PC100 / DCX AT100 controllers only).
MC_STAT_PJOG_ON	Selects the Plus Jog On status bit (DCX PC100 / DCX AT100 controllers only).
MC_STAT_PLIM_ENAB	Selects the Plus Hard Limit Enable status bit.
MC_STAT_PLIM_TRIP	Selects the Plus Hard Limit Tripped status bit.
MC_STAT_POS_CAPT	Selects the Position Captured status bit (DCX PCI300 controllers only).
MC_STAT_PROG_DIR	Selects the Programmed Direction status bit (DC2 PC100 controllers only).
MC_STAT_PSOFT_ENAB	Selects the Plus Soft Limit Enable status bit (DCX AT200, DCX AT300, DCX PCI controllers only).
MC_STAT_PSOFT_TRIP	Selects the Plus Soft Limit Tripped status bit (DCX AT200, DCX AT300, DCX PCI controllers only).
MC_STAT_RECORD	Selects the Position status bit (DC2 PC100 controllers only).
MC_STAT_STOPPING	Selects the Stopping status bit (DC2 PC100 controllers only).
MC_STAT_SYNC	Selects the Synchronize status bit (DC2 PC100 controllers only).
MC_STAT_TRAJ	Selects the Trajectory Complete status bit.
MC_STEP_FULL	Selects stepper motor full step operation.
MC_STEP_HALF	Selects stepper motor half step operation.
MC_TYPE_DOUBLE	Used with register get/set functions to select a double precision floating point data type.
MC_TYPE_FLOAT	Used with **pmccmdex( )** and register get/set functions to select a single precision floating point data type.
MC_TYPE_LONG	Used with register get/set functions to select a long integer (32-bit) data type.
MC_TYPE_NONE	Used with **pmccmdex( )** to specify no argument.
MC_TYPE_REG	Used with **pmccmdex( )** to select a register based argument.
MC_TYPE_SERVO	Indicates the axis is a servo motor – used with the **MCAXISCONFIG** structure.
MC_TYPE_STEPPER	Indicates the axis is a stepper motor – used with the **MCAXISCONFIG** structure.
MC_TYPE_STRING	Used with **pmccmdex( )** and register get/set functions to select a string data type.
MC100	Identifies a DC Servo axis with analog signal output.

Constant	Description
MC110	Identifies a DC Servo axis with motor output.
MC150	Identifies a stepper motor axis.
MC160	Identifies a stepper motor with encoder axis.
MC200	Identifies an Advanced Servo axis with analog signal output.
MC210	Identifies an Advanced Servo axis with PWM motor output.
MC260	Identifies an Advanced Stepper axis.
MC300	Identifies a DSP-Based Servo axis with analog signal output.
MC302	Identifies a DSP-Based Dual Servo axes with dual analog signal outputs.
MC320	Identifies a DSP-Based Brushless-AC Servo axis with analog signal output.
MC360	Identifies a DSP-Based Stepper axis.
MC362	Identifies a DSP-Based Dual Stepper axes.
MC400	Identifies this axis as providing additional digital I/O channels (16).
MC500	Identifies this axis as providing additional analog channels.
MCERR_ALL_AXES	Error code indicating you may not use the constant MC_ALL_AXES with this function.
MCERR_ALLOC_MEM	There was a memory allocation error during a call to **MCOpen( )**. Try closing other Windows programs to free memory.
MCERR_AXIS_NUMBER	Error code indicating that the specified axis number is out of range.
MCERR_AXIS_TYPE	Error code indicating that the function does not apply to the axis specified.
MCERR_COMM_PORT	Error code indicating and invalid constant value was given as the argument to a function.
MCERR_CONSTANT	Error code indicating and invalid constant value was given as the argument to a function.
MCERR_CONTROLLER	Error code indicating the controller handle is invalid.
MCERR_INIT_DRIVER	**MCOpen( )** was unable to initialize the device driver for this controller.
MCERR_MODE_UNAVAIL	The requested open mode for **MCOpen( )** was unavailable. This can occur when a non-multitasking controller is already open in a mode that is different from the requested mode.
MCERR_NO_CONTROLLER	Returned by **MCOpen( )** when no controller has been configured for this ID number.
MCERR_NO_REPLY	Error code indicating a controller failed to reply.
MCERR_NOERROR	Error code return value indicating that no errors have occurred.
MCERR_NOT_FOUND	Restore operation could not find data.
MCERR_NOT_INITIALIZED	An attempt was made to use a controller feature before that feature had been initialized.
MCERR_NOT_PRESENT	The controller hardware was not found during a call to **MCOpen( )**. Check the MCAPI settings with the setup program.
MCERR_NOTSUPPORTED	Error code indicating function is not supported by this controller. The MCAPI will handle this condition by ignoring requests to set this parameter and by returning a fixed default value for the parameter. You may, therefore, safely ignore this error.
MCERR_OBSOLETE	Error code indicating function is obsolete. See manual for updated function.

Constant	Description
MCERR_OPEN_EXCLUSIVE	Returned by **MCOpen( )** when it is unable to satisfy a request for an exclusive handle. You cannot obtain an exclusive handle to a controller if there are other open handles for the controller at the time of your request.
MCERR_OUT_OF_HANDLES	Returned by **MCOpen( )** when the device driver has no more free handles it can assign to this request.
MCERR_RANGE	Error code indicating a parameter was out of range.
MCERR_REPLY_AXIS	Error code indicating the wrong axis number replied to a function.
MCERR_REPLY_COMMAND	Error code indicating the controller reply does not match the command.
MCERR_REPLY_SIZE	Error code indicating the length of a reply was incorrect (too many or too few bytes).
MCERR_TIMEOUT	A timeout occurred while attempting to send a command or read a reply from the controller.
MCERR_UNKNOWN_REPLY	Error code indicating an unknown or unexpected reply was received from a controller.
MCERR_UNSUPPORTED_MODE	Return value from **MCOpen( )** when the requested mode is not supported for this controller/interface combination.
MCERR_WINDOW	Error code indicating a window handle is invalid.
MCERRMASK_AXIS	Error mask value for **MCErrorNotify( )** to enable error messages for out of range axis numbers and invalid usage of MC_ALL_AXES.
MCERRMASK_HANDLE	Error mask value for **MCErrorNotify( )** to enable error messages for invalid controller or window handles.
MCERRMASK_IO	Error mask value for **MCErrorNotify( )** to enable error messages for controller communication errors.
MCERRMASK_PARAMETER	Error mask value for **MCErrorNotify( )** to enable error messages for invalid or out of range parameters to MCAPI functions.
MCERRMASK_STANDARD	Collection of most common error mask values for **MCErrorNotify( )** (includes all errors except MCERRMASK_UNSUPPORTED) .
MCERRMASK_UNSUPPORTED	Error mask value for **MCErrorNotify( )** that enables error notification when a function is called that is not supported by the controller.
MF300	Identifies this axis as an RS-232 communications module. This module is not normally used with a controller installed in a PC adapter slot.
MF310	Identifies this axis as an IEEE-488 (GPIB) communications module. This module is not normally used with a controller installed in a PC adapter slot.
NO_CONTROLLER	One setting for the **ControllerType** member of an **MCPARAMEX** structure, it indicates that no controller is installed at this ID.
NO_MODULE	Identifies this axis as having no module installed.
NONE	One setting for the **ControllerType** member of a **MCPARAMEX** structure, it indicates that no controller is installed at this ID. This is an old constant - it is recommended that you use NO_CONTROLLER instead of NONE.

Chapter Contents

Chapter

22

MCAPI Status Word Constants Lookup Table

This table is provided for cross-platform comparisons of **MCDecodeStatus( )** constants. Suppose you are using the MC_STAT_TRAJ status bit on a DC2-PC100 controller and plan to migrate to the more powerful DCX-PCI300 controller. Locate the constant in the leftmost column, read across the row to the DCX-PCI300 column and you will see that the MC_STAT_TRAJ constant is also supported for the DCX-PCI300.

You will also notice that the bit positions for MC_STAT_TRAJ on the DC2-PC100 and the DCX-PCI300 are different. If you had hard-coded this bit in your application, you would be forced to change your program to accommodate a different controller. By using **MCDecodeStatus( )** and the appropriate constants, no changes are required!

The numbers in the table represent the status word bit position for the specific controller. A dash indicates the constant is not supported for a particular controller.

Bit	DC2-PC DC2-STN	DCX-PC100 DCX-AT100	DCX-AT200 DCX-AT300	DCX-PCI100 DCX-PCI300
0	MC_STAT_MTR_ENABLE	MC_STAT_BUSY	MC_STAT_BUSY	MC_STAT_BUSY
1	MC_STAT_ERROR	MC_STAT_MTR_ENABLE	MC_STAT_MTR_ENABLE	MC_STAT_MTR_ENABLE
2	MC_STAT_CAPTURE	MC_STAT_MODE_VEL	MC_STAT_AT_TARGET	MC_STAT_AT_TARGET
3	MC_STAT_BREAKPOINT	MC_STAT_TRAJ	MC_STAT_TRAJ	MC_STAT_TRAJ
4	MC_STAT_TRAJ	MC_STAT_DIR	MC_STAT_DIR	MC_STAT_DIR
5	MC_STAT_STOPPING	MC_STAT_PHASE	MC_STAT_JOG_ENAB	- NONE -
6	- NONE-	MC_STAT_HOMED	MC_STAT_HOMED	MC_STAT_HOMED
7	MC_STAT_DIR	MC_STAT_ERROR	MC_STAT_ERROR	MC_STAT_ERROR
8	MC_STAT_AT_TARGET	MC_STAT_LOOK_INDEX	MC_STAT_LOOK_INDEX	MC_STAT_LOOK_INDEX
9	MC_STAT_PHASE	MC_STAT_LOOK_EDGE	MC_STAT_LOOK_EDGE	MC_STAT_LOOK_EDGE
10	MC_STAT_LOOK_INDEX	MC_STAT_FULL_STEP	- NONE-	MC_STAT_INDEX_FOUND
11	MC_STAT_LOOK_EDGE	MC_STAT_HALF_STEP	- NONE -	MC_STAT_POS_CAPT
12	MC_STAT_HOMED	MC_STAT_BREAKPOINT	MC_STAT_BREAKPOINT	MC_STAT_BREAKPOINT
13	MC_STAT_INP_HOME	MC_STAT_JOGGING	MC_STAT_FOLLOWING	MC_STAT_FOLLOWING
14	MC_STAT_RECORD	MC_STAT_AMP_ENABLE	MC_STAT_AMP_ENABLE	MC_STAT_AMP_ENABLE
15	MC_STAT_SYNC	MC_STAT_AMP_FAULT	MC_STAT_AMP_FAULT	MC_STAT_AMP_FAULT
16	MC_STAT_ACCEL	MC_STAT_PLIM_ENAB	MC_STAT_PLIM_ENAB	MC_STAT_PLIM_ENAB
17	MC_STAT_MODE_POS	MC_STAT_PLIM_TRIP	MC_STAT_PLIM_TRIP	MC_STAT_PLIM_TRIP
18	MC_STAT_MODE_VEL	MC_STAT_MLIM_ENAB	MC_STAT_MLIM_ENAB	MC_STAT_MLIM_ENAB
19	MC_STAT_MODE_TRQE	MC_STAT_MLIM_TRIP	MC_STAT_MLIM_TRIP	MC_STAT_MLIM_TRIP
20	MC_STAT_MODE_ARC	MC_STAT_PJOG_ENAB	MC_STAT_PSOFT_ENAB	MC_STAT_PSOFT_ENAB
21	MC_STAT_MODE_CNTR	MC_STAT_PJOG_ON	MC_STAT_PSOFT_TRIP	MC_STAT_PSOFT_TRIP
22	MC_STAT_MODE_SLAVE	MC_STAT_MJOG_ENAB	MC_STAT_MSOFT_ENAB	MC_STAT_MSOFT_ENAB
23	MC_STAT_MODE_LIN	MC_STAT_MJOG_ON	MC_STAT_MSOFT_TRIP	MC_STAT_MSOFT_TRIP
24	MC_STAT_LMT_ABORT	MC_STAT_INP_INDEX	MC_STAT_INP_INDEX	MC_STAT_INP_INDEX
25	MC_STAT_LMT_STOP	MC_STAT_INP_HOME	MC_STAT_INP_HOME	MC_STAT_INP_HOME
26	MC_STAT_MLIM_TRIP	MC_STAT_INP_AMP	MC_STAT_INP_AMP	MC_STAT_INP_AMP
27	MC_STAT_MLIM_ENAB	- NONE -	MC_STAT_INP_AUX	MC_STAT_INP_AUX
28	MC_STAT_INP_MLIM	MC_STAT_INP_PLIM	MC_STAT_INP_PLIM	MC_STAT_INP_PLIM
29	MC_STAT_PLIM_TRIP	MC_STAT_INP_MLIM	MC_STAT_INP_MLIM	STAT_INP_MLIM
30	MC_STAT_PLIM_ENAB	MC_STAT_INP_PJOG	MC_STAT_INP_USER1	MC_STAT_INP_NULL
31	MC_STAT_INP_PLIM	MC_STAT_INP_MJOG	MC_STAT_INP_USER2	- NONE -

Chapter Contents

Chapter 23

Motion Dialog Windows Classes

The motion dialog window classes supplement the motion dialog functions to provide the programmer simple and effective tools to build attractive graphical user interfaces.

MCDLG_LEDCLASS

#include "mcdlg.h"

Creates a window with a small graphical LED and text label to the right of it. The LED window class is based on the checkbox style windows BUTTON class. To change the color of the LED send it a BM_SETCHECK message with a WPARAM of BST_CHECKED for the on color (default green), BST_UNCHECKED for the off color (default dark gray), or BST_INDETERMINATE for the error color (default red).

LED CLASS Styles

The LED class responds to the standard window styles (WS_xxx) and button styles (BS_xxx) applicable to checkbox windows. Use BS_LEFTTEXT to locate the text to the left of the LED graphic.

LED CLASS Messages

LEDM_GETCHECKCOLOR

Returns the current color of the "Checked" (on) state for the LED as a COLORREF.

```
   wParam = (WPARAM) 0;          // unused, must be 0
   lParam = (LPARAM) 0;          // unused, must be 0
```

LEDM_GETUNCHECKCOLOR

Returns the current color of the "Unchecked" (off) state for the LED as a COLORREF.

```
   wParam = (WPARAM) 0;          // unused, must be 0
   lParam = (LPARAM) 0;          // unused, must be 0
```

LEDM_GETINDETRMCOLOR

Returns the current color of the "Indeterminate" state for the LED as a COLORREF.

```
   wParam = (WPARAM) 0;          // unused, must be 0
   lParam = (LPARAM) 0;          // unused, must be 0
```

LEDM_SETCHECKCOLOR

Sets the color of the "Checked" (on) state for the LED. By default this color is bright green - RGB(0, 255, 0).

```
   wParam = (WPARAM) 0;          // TRUE to force an immediate redraw
   lParam = (LPARAM) rgbColor;   // COLORREF color value
```

LEDM_SETUNCHECKCOLOR

Sets the color of the "Unchecked" (off) state for the LED.

```
   wParam = (WPARAM) 0;          // TRUE to force an immediate redraw
   lParam = (LPARAM) rgbColor;   // COLORREF color value
```

LEDM_SETINDETRMCOLOR

Sets the color of the "Indeterminate" state for the LED. By default this color is bright red - RGB(255, 0, 0).

```
   wParam = (WPARAM) 0;          // TRUE to force an immediate redraw
   lParam = (LPARAM) rgbColor;   // COLORREF color value
```

MCDLG_READOUTCLASS

#include "mcdlg.h"

Creates a single line "readout" window, similar to a text box. By default the text is green on a black background, and the window font is scaled to the window size to make it easy to create large readouts. The READOUT window class is based on the Windows STATIC class. To change the displayed text of the READOUT the standard WM_SETTEXT message may be sent to the window.

READOUT CLASS Styles

The READOUT class responds to the standard window styles (WS_xxx) and static styles (SS_xxx) applicable to static windows. Use RDTS_LEFT, RDTS_CENTER, or RDTS_RIGHT to set the justification of the text within the window.

When you declare a READOUT in a dialog box template using the CONTROL statement the dialog box manager will set the READOUT font to the default dialog box font. This can lead to undesirable behavior (i.e. the wrong size font). The READOUT class normally responds to the WM_SETFONT message (which is what the dialog box manager sends to mess things up), however if you specify the RDTS_DIALOGBOX style when creating the READOUT window it will ignore WM_SETFONT messages. See the CWDEMO sample program for an example.

READOUT CLASS Messages

RDTM_GETTEXTCOLOR

Returns the current color of the readout text (default green) as a COLORREF.

```
   wParam = (WPARAM) 0;          // unused, must be 0
   lParam = (LPARAM) 0;          // unused, must be 0
```

RDTM_GETBKCOLOR

Returns the current color of the readout background (default black) as a COLORREF.

```
   wParam = (WPARAM) 0;          // unused, must be 0
   lParam = (LPARAM) 0;          // unused, must be 0
```

RDTM_SETTEXTCOLOR

Sets the color of the readout text.

```
   wParam = (WPARAM) 0;          // TRUE to force an immediate redraw
   lParam = (LPARAM) rgbColor;   // COLORREF color value
```

RDTM_SETBKCOLOR

Sets the color of the readout background.

```
   wParam = (WPARAM) 0;          // TRUE to force an immediate redraw
   lParam = (LPARAM) rgbColor;   // COLORREF color value
```

Chapter Contents

- Motherboard: DCX-PCI100
- DCX-MC100 - +/- 10 Volt Analog Servo Motor Control Module
- DCX-MC110 - Direct Motor Drive Servo Control Module
- DCX-MC400 - 16 channel Digital I/O Module
- DCX-MC5X0 - Analog I/O Module

Chapter 24

DCX Specifications

Motherboard: DCX-PCI100

Function	8 Axis Motion Controller
Installation	Intel PC compatible computer
Configuration	8 User Installed Modules
Main Processor	QED 5231 200MHz MIPS RISC
Processor Clock	192 MHz
Memory	512k x 8 bit Flash Memory
	1Meg X 32 Synchronous Dynamic Ram
Processor Fault Detection	Watchdog Circuit with Reset Relay
Status LED's	Power, Reset, Run, (8) Motor Error
Standard Communication Interface	PCI Bus 4 Kilobytes dual ported memory in Memory Address Space 'Plug and Play' dynamic addressing
Undedicated Digital I/O Channels	16 TTL (0 – 5 VDC), 1ma max. sink/source, 4.7K ohm pull up to +5V 2 groups (8 inputs, 8 outputs)
Required Supply Voltages	+5,+12 and -12 vdc
Form Factor	Full Size PCI card (4.2" x 12.28")
Operating Temperature range	0 degrees C to 60 degrees C
Weight	10 oz + 1.2 oz per module (approx.)

DCX-MC100 - +/- 10 Volt Analog Servo Motor Control Module

Function	Closed Loop Servo Controller
Installation	DCX-PCI100 Motion Control Motherboard
Operating Modes	Position, Velocity
Filter Algorithm	PID
Filter Update Rate	2.932 KHz
Trajectory Generator	Trapezoidal with common Acceleration / Deceleration
Position Feedback	Incremental Encoder with Index
Position and Velocity Resolution	30 bit
Output	Analog Signal (+/- 10 vdc @ 10 ma, 12 bit)
Encoder and Index Inputs	Differential or single ended, -7 to +7 vdc max.
Encoder Count Rate	750,000 Quadrature Counts/Sec.
Encoder Supply Voltage	+5 or +12 vdc, user selectable
Axis Inputs	Limit+, Limit-, Coarse Home, Amplifier Fault (TTL level, low active)
Axis Outputs	Amplifier Inhibit (TTL compatible)
Operating Temperature range	0 degrees C to 60 degrees C

DCX-MC110 – Direct Drive Servo Control Module

Function	Closed Loop Servo Controller
Installation	DCX-PCI100 Motion Control Motherboard
Operating Modes	Position, Velocity
Filter Algorithm	PID
Filter Update Rate	2.932 KHz
Trajectory Generator	Trapezoidal with common Acceleration / Deceleration
Position Feedback	Incremental Encoder with Index
Position and Velocity Resolution	30 bit
Output	0 - +12 volt @ 0.5A
Encoder and Index Inputs	Differential or single ended, -7 to +7 vdc max.
Encoder Count Rate	750,000 Quadrature Counts/Sec.
Encoder Supply Voltage	+5 or +12 vdc, user selectable
Axis Inputs	Limit+, Limit-, Coarse Home, Amplifier Fault (TTL level, low active
Axis Outputs	Amplifier Inhibit (TTL compatible)
Operating Temperature range	0 degrees C to 60 degrees C

DCX-MC400 - 16 channel Digital I/O Module

Function	16 Channel Digital I/O module
Installation	DCX-PCI100 Motion Control Motherboard
Channels	16, individually programmable as inputs or outputs
Output low voltage (min)	0.0 volt
Output high voltage (min)	2.4 volt
Current sink	1 ma max.
Current source	1 ma max.
Input Low voltage	-0.3V min. to 0.8V max.
Input High voltage	2.0V min. to 5.3V max.
Input termination	4.7K ohm per channel
Relay rack interface	DCX-BF022
Operating Temperature range	0 degrees C to 60 degrees C

DCX-MC5X0 - Analog I/O Module

Function	DCX-MC500 – 4 A/D channels, 4 D/A channels DCX-MC510 – 4 A/D channels DCX-MC520 – 4 D/A channels
Installation	DCX-PCI100 Motion Control Motherboard
Input resolution	12 bit
Input voltage range	0.0V to +5.0V
Output resolution	12 bit
Output voltage range	0.0V to +5.0V (@ 5ma), -10V to +10V (@ 5ma)
Output Offset Adjustment	20 turn trim pot (+/- 10V outputs only)
Output Full Scale Adjustment	single turn trim pot (+/- 10V outputs only)
Operating Temperature range	0 degrees C to 60 degrees C

DCX-MC500 Electrical Specifications

Parameter	Min.	Max	Unit
Input Resolution	12		Bits
Input Conversion Rate		10	KHz
Input Zero Error Using Internal Reference Using External Reference		 +/- 3 +/- 1/2	 LSB LSB
Input Full-Scale Error Using Internal Reference Using External Reference		 +/- 15 +/- 1/2	 LSB LSB
Input Zero Temp. Coefficient		0.5	ppm/C
Input Differential Nonlinearity		+/- 1	LSB
Input Total Unadjusted Error Using Internal Reference Using External Reference		 +/- 15 +/- 1	
Input Voltage Range Using Internal Reference Using External Reference	 0.0 0.0	 5.0 Vref	
Input Capacitance		8	
Input Leakage Current		100	
External Reference Voltage	4.0	6.0	

Parameter	Min.	Max	Unit
Output Resolution	12		Bits
Output Zero Code Error *			LSB
Output Full Scale Error *			LSB
Output Nonlinearity *			LSB
Output Total Unadjusted Error *			LSB
Output Voltage Range	0.0 -10.0	5.0 +10.0	V V

* These values are for 0 to +5.0 volt outputs

Chapter Contents

Chapter 25

Connectors, Jumpers, and Schematics

DCX-PCI100 Motion Control Motherboard

Status LED Indicators

LED #	*Color*	*Description*
D1	Green	+5V logic supply
D2	Yellow	DCX Reset active
D3	Green	Run (processor fault or watchdog tripped if off)
L1	Red	Module #1 motor error (exceeded max. following error or limit tripped)
L2	Red	Module #2 motor error (exceeded max. following error or limit tripped)
L3	Red	Module #3 motor error (exceeded max. following error or limit tripped)
L4	Red	Module #4 motor error (exceeded max. following error or limit tripped)
L5	Red	Module #5 motor error (exceeded max. following error or limit tripped)
L6	Red	Module #6 motor error (exceeded max. following error or limit tripped)
L7	Red	Module #7 motor error (exceeded max. following error or limit tripped)
L8	Red	Module #8 motor error (exceeded max. following error or limit tripped)

(Refer to diagram at the end of this appendix)

General Purpose I/O (Digital I/O and Analog inputs) Connector J5

Pin #	*Description*
1	+5 VDC
2	RESET RELAY CONTACT #1 *
3	DIGITAL OUTPUT CHANNEL 16
4	RESET RELAY CONTACT #2 *
5	DIGITAL OUTPUT, CHANNEL 15
6	DIGITAL OUTPUT, CHANNEL 14
7	DIGITAL OUTPUT, CHANNEL 13
8	DIGITAL OUTPUT, CHANNEL 12
9	DIGITAL OUTPUT, CHANNEL 11
10	DIGITAL OUTPUT, CHANNEL 10
11	DIGITAL OUTPUT, CHANNEL 09
12	DIGITAL INPUT, CHANNEL 08
13	DIGITAL INPUT, CHANNEL 07
14	DIGITAL INPUT, CHANNEL 06
15	DIGITAL INPUT, CHANNEL 05
16	DIGITAL INPUT, CHANNEL 04
17	DIGITAL INPUT, CHANNEL 03
18	DIGITAL INPUT, CHANNEL 02
19	DIGITAL INPUT, CHANNEL 01
20	NO CONNECT
21	+12 VDC
22	NO CONNECT
23	NO CONNECT
24	GROUND
25	-12 VDC
26	GROUND

Mating Connector:26-pin dual-row IDC female, Circuit Assembly P/N 26IDS2-C-SPT-SR or equivalent

*** -** Reset Relay contacts (normally open). The relay is energized (contacts 1 and 2 connected) when the DCX-PCI100 is held in reset.

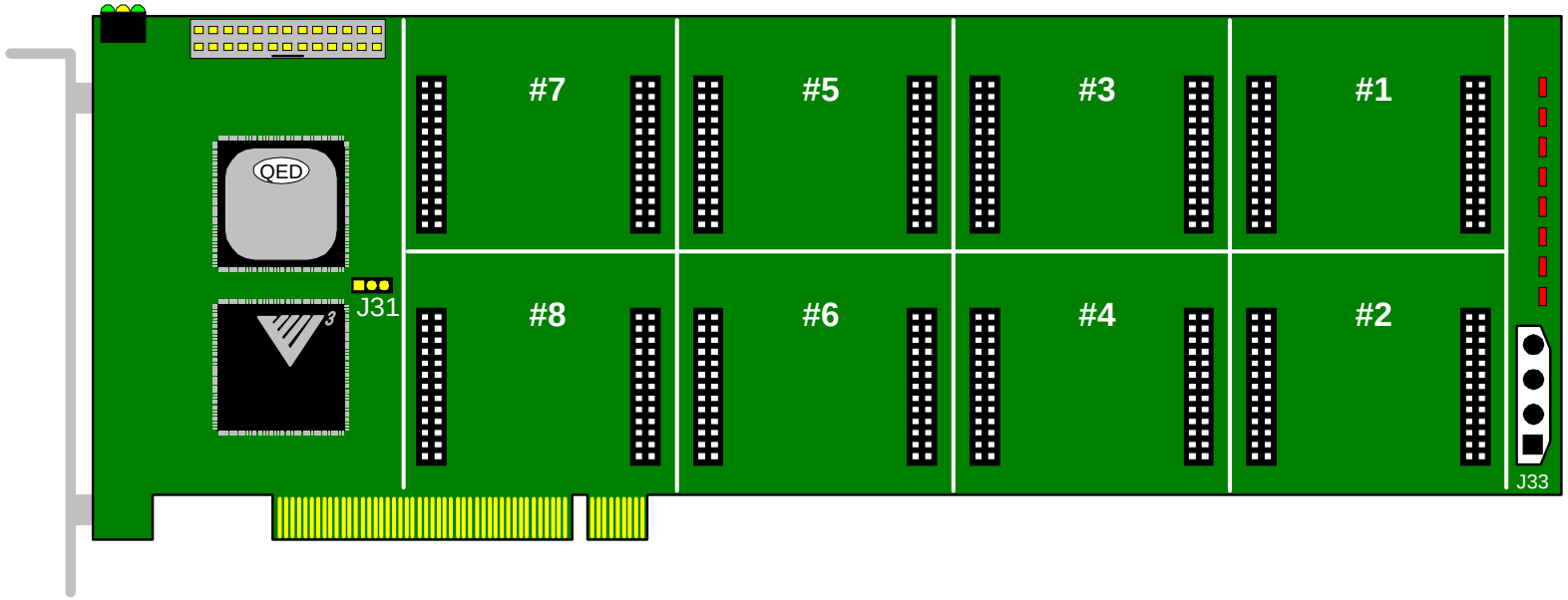
J31 – +12 volt supply input select

Pins	*Description*
Open	**+12 volt supply provided via connector J33**
2 to 3	Consult factory

J33 - +12 volt Motor Supply connector

Pin #	*Description*
1	+12 volt input (identified by square pad on bottom side of PCB)
2	No connect
3	Ground
4	No Connect

Mating Connector: PMC P/N = 71.060.A (Disk Drive Power Cable Splitter)
Newark Electronics P/N = 83F7055 (GC Electronics P/N 45-0104)



DCX-PCI100 Motion Control Motherboard

DCX-MC100 +/- 10V Servo Motor Control Module

SIGNAL DESCRIPTIONS:

Analog Command Return
connection point: J3 - pin 1
signal type: ground
notes:
explanation: Provides the signal ground for the modules Analog Command Signal output. This return path is common to the ground plane of the DCX motherboard, but is connected in such a way as to reduce digital noise. Typical servo amplifiers will have a connection for the analog command return where this signal should be connected.

Analog Command Output
connection point: J3 - pin 2
signal type: +/- 10V analog, 12 bit
notes: connects to servo amplifier motor command input
explanation: This module output signal is used to control the servo amplifier's output. When connected to the command input of a velocity mode amplifier, the voltage level on this signal should cause the amplifier to drive the servo at a proportional velocity. For current mode amplifiers, the voltage level should cause a proportional current to be supplied to the servo. The module provides an analog signal that is in the range -10 to +10 volts, with 0 volts being the null output level. Positive voltages indicate a desired velocity or current in one direction. Negative voltages indicate velocity or current in the opposite direction. The maximum drive current of this signal is +/-10 milliamps.

Coarse Home Input
connection point: J3 - pin 9
signal type: TTL input
notes: 4.7K pull up resistor is connected to the +5V logic supply
explanation: This module input is used to determine the proper zero position of the servo. In servo systems that use rotary encoders with index outputs, an index pulse is generated once per rotation of the encoder. While this signal occurs at a very repeatable angular position on the encoder, it may occur many times within the motion range of the servo. In these cases, a Coarse Home switch connected to this module input can be used to qualify which index pulse is the true zero position of the servo. By setting this switch to be activated near the end of travel of the servo, and using DCX motion commands to position the servo within this region prior to searching for the index pulse, a unique zero position for the servo can be determined.

Amplifier Fault Input
connection point: J3 - pin 10
signal type: TTL input
notes: 4.7K pull up resistor is connected to the +5V logic supply
explanation: - This module input is designed to be connected to the servo amplifiers Fault or Error output signal. The state of this signal will appear as a status bit in the servo's status word. Using the Fault oN command, this signal can be enabled to shut the axis off if the input goes active low. In this condition, no further servo motion will occur until the fail signal is deactivated and the Motor oN command is issued. The Fault oFf command can be used to disable this signal.

Amplifier Inhibit
connection point: J3 - pin 11
signal type: TTL output
notes: 2ma sink/source
explanation: - This module output signal should be connected to the enable input of the servo amplifier. When the DCX is turned on or reset, this signal will immediately go to its' inactive high level. When the Motor oN command is issued to the DCX, this signal will go to its' active low level. Anytime there is an error on the respective servo axis, including ***exceeding the following error, a limit switch input activated or the Amplifier Fault input activated***, the Amplifier Inhibit signal will be activated. This signal can also be deactivated by the Motor oFf command.

Limit Positive and Limit Negative Inputs
connection point: J3 - pin 14 (Limit Positive), J3 - pin 15 (Limit Negative)
signal type: TTL input
notes: 4.7K pull up resistor is connected to the +5V logic supply
explanation: The limit switch inputs are used to cause the DCX to stop a servo's motion when it reaches the end of travel. If the servo is in position mode, the axis will only be stopped if it is moving in the direction of an activated limit switch. In all other modes, the servo will be stopped regardless of the direction it is moving if either limit switch is activated. There are three modes of stopping that can be configured by the Limit Mode command. The limit switch inputs can be enabled and disabled with the Limits oN and Limits oFf commands respectively. See the description of **Motion Limits** in the **Motion Control** chapter.

Primary Encoder Inputs (Phase A+, Phase -, Phase B+, Phase B-, Index+, Index-)
connection point: see pin-out table
signal type: TTL or Differential driver output (-7V to +7V)
notes:
explanation: These input signals should be connected to an incremental quadrature encoder for supplying position feedback information for the servo controller. The plus (+) and minus (-) signs refer to the two sides of differential inputs. The default shipping configuration is for single ended encoders. When a differential encoder is used the signal trace (on the back side of the module) between Jp2 and Jp3 must be cut.

Encoder Power Output
connection point: J3 pin 17
signal type: +5 VDC PC power supply output or +12 VDC PC power supply output
notes:
explanation: This module pin provides a convenient supply voltage connection for the encoders. The module is shipped configured for +5 volt encoder supply. To configure the module for a +12 volt encoder, cut the signal on the back side of the module between JP4 pins 2 & 3.

SUPPLY CONNECTIONS (+5, +12, -12, GROUND) - These module pins provide access to the DCX supply voltages.

DCX-MC100 Module connectors

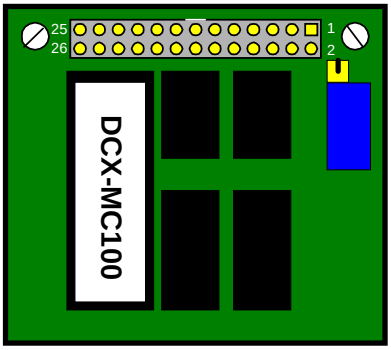
J3 connector pin-out (Motor command, encoders, and axis I/O)

Pin #	Description
1	Analog Command return (analog ground)
2	Analog Command output (output, +/-10 V)
3	+12 VDC
4	-12 VDC
5	Ground
6	+5 VDC
7	Reserved
8	Reserved
9	Coarse Home (input, active low, with 4.7K ohm pull-up to +5V)
10	Amplifier Fault (input, active low, with 4.7K ohm pull-up to +5V)
11	Amplifier Inhibit (output, active low, TTL level)**
12	Reserved
13	Reserved
14	Limit Positive (input, active low, with 4.7K ohm pull-up to +5V)
15	Limit Negative (input, active low, with 4.7K ohm pull-up to +5V)
16	Encoder Phase B+ (input)*
17	Encoder Power (+5VDC or +12VDC, see jumper JP3)
18	Ground
19	Encoder Phase B- (input)
20	Encoder Phase A- (input)
21	Ground
22	Encoder Power
23	Encoder Phase A+ (input)*
24	Encoder Power
25	Encoder Index- (input, active low)
26	Ground

* Use A+ and B+ for single-ended ENCODER INPUTS
** These signals are not suitable for directly driving optically isolated inputs.

Mating Connector:26-pin dual-row IDC female, Circuit Assembly P/N 26IDS2-C-SPT-SR or equivalent

DCX MODULE CONNECTOR PIN NUMBERING
(TOP SIDE VIEW)



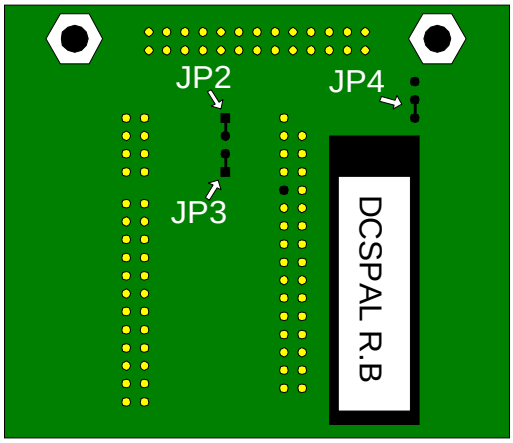
DCX-MC100 Module Configuration Jumpers - configuration in **bold type** denotes default factory shipping configuration

JP2 & JP3 – Encoder type (Single ended versus differential)

Pins	***Description***
1 to 2	**Single ended encoder A, B (pcb trace)**
	Differential encoder A+, A-, B+, B- (cut JP2 and JP3 pcb trace)

JP4 – Encoder Power Select (+5VDC or +12 VDC)

Pins	***Description***
1 to 2	+12 VDC encoder supply on J3 pin 17 (and cut pcb trace from JP4-2 to JP4-3)
2 to 3	**+5 VDC encoder supply on J3 pin 17 (pcb trace)**



DCX-MC100 Module Output Offset Potentiometers

This multi-turn trimming potentiometer can be used to add an offset to the module's analog output. The range of this adjustment is approximately +/-1.0 volts.

DCX-MC110 Motor Drive Servo Control Module

SIGNAL DESCRIPTIONS:

Motor Drive Outputs
connection point: J3 - pin 1 (Motor Drive +), J3 – pin 6 (Motor Drive -)
signal type: 8 bit Analog, 0 to +12 volts @ 0.5A
notes:
explanation: These module outputs provide the direct motor drive for a DC servo motor. The resolution of the motor drive outputs are eight bits. Rotational direction is determined by connecting the Motor Drive signals (Motor - and Motor +) to the appropriate terminals on the DC servo motor.

Positive Supply Input
connection point: J3 - pin 7
signal type: Optional power supply input
notes:
explanation: Consult factory for details

Negative Supply Input
connection point: J3 - pin 8
signal type: Optional power supply input
notes:
explanation: Consult factory for details

Coarse Home Input
connection point: J3 - pin 9
signal type: TTL input
notes: 4.7K pull up resistor is connected to the +5V logic supply
explanation: This module input is used to determine the proper zero position of the servo. In servo systems that use rotary encoders with index outputs, an index pulse is generated once per rotation of the encoder. While this signal occurs at a very repeatable angular position on the encoder, it may occur many times within the motion range of the servo. In these cases, a Coarse Home switch connected to this module input can be used to qualify which index pulse is the true zero position of the servo. By setting this switch to be activated near the end of travel of the servo, and using DCX motion commands to position the servo within this region prior to searching for the index pulse, a unique zero position for the servo can be determined.

Amplifier Fault Input
connection point: J3 - pin 10
signal type: TTL input
notes: 4.7K pull up resistor is connected to the +5V logic supply
explanation: - This module input is designed to be connected to the servo amplifiers Fault or Error output signal. The state of this signal will appear as a status bit in the servo's status word. Using the Fault oN command, this signal can be enabled to shut the axis off if the input goes active low. In this condition, no further servo motion will occur until the fail signal is deactivated and the Motor oN command is issued. The Fault oFf command can be used to disable this signal.

Amplifier Inhibit
connection point: J3 - pin 11
signal type: TTL output
notes: 2ma sink/source
explanation: - This module output signal should be connected to the enable input of the servo amplifier. When the DCX is turned on or reset, this signal will immediately go to its' inactive high level. When the Motor oN command is issued to the DCX, this signal will go to its' active low level. Anytime there is an error on the respective servo axis, including ***exceeding the following error, a limit switch input activated or the Amplifier Fault input activated***, the Amplifier Inhibit signal will be activated. This signal can also be deactivated by the Motor oFf command.

Limit Positive and Limit Negative Inputs
connection point: J3 - pin 14 (Limit Positive), J3 - pin 15 (Limit Negative)
signal type: TTL input
notes: 4.7K pull up resistor is connected to the +5V logic supply
explanation: The limit switch inputs are used to cause the DCX to stop a servo's motion when it reaches the end of travel. If the servo is in position mode, the axis will only be stopped if it is moving in the direction of an activated limit switch. In all other modes, the servo will be stopped regardless of the direction it is moving if either limit switch is activated. There are three modes of stopping that can be configured by the Limit Mode command. The limit switch inputs can be enabled and disabled with the Limits oN and Limits oFf commands respectively. See the description of **Motion Limits** in the **Motion Control** chapter.

Primary Encoder Inputs (Phase A+, Phase -, Phase B+, Phase B-, Index+, Index-)
connection point: see pin-out table
signal type: TTL or Differential driver output (-7V to +7V)
notes:
explanation: These input signals should be connected to an incremental quadrature encoder for supplying position feedback information for the servo controller. The plus (+) and minus (-) signs refer to the two sides of differential inputs. The default shipping configuration is for single ended encoders. When a differential encoder is used the signal trace (on the back side of the module) between Jp2 and Jp3 must be cut.

Encoder Power Output
connection point: J3 pin 17
signal type: +5 VDC PC power supply output or +12 VDC PC power supply output
notes:
explanation: This module pin provides a convenient supply voltage connection for the encoders. The module is shipped configured for +5 volt encoder supply. To configure the module for a +12 volt encoder, cut the signal on the back side of the module between JP4 pins 2 & 3.

SUPPLY CONNECTIONS (+5, +12, -12, GROUND) - These module pins provide access to the DCX supply voltages.

DCX-MC110 Module connectors

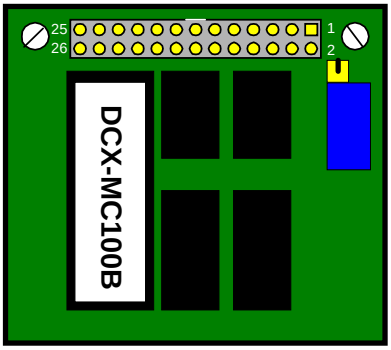
J3 connector pin-out (Motor command, encoders, and axis I/O)

Pin #	*Description*
1	Motor Drive + (output, 500ma max.)
2	Encoder Power (+5VDC or +12VDC, see jumper JP4)
3	Encoder Phase A+ (input)*
4	Encoder Phase B+ (input)*
5	Ground
6	Motor Drive - (output, 500ma max.)
7	Positive Supply (input, optional, consult factory) ***
8	Negative Supply (input, optional, consult factory) ***
9	Coarse Home (input, active low, with 4.7K ohm pull-up to +5V)
10	Amplifier Fault (input, active low, with 4.7K ohm pull-up to +5V)
11	Amplifier Inhibit (output, active low, TTL level)**
12	Reserved
13	Reserved
14	Limit Positive (input, active low, with 4.7K ohm pull-up to +5V)
15	Limit Negative (input, active low, with 4.7K ohm pull-up to +5V)
16	Encoder Phase B+ (input)*
17	Encoder Power (+5VDC or +12VDC, see jumper JP4)
18	Ground
19	Encoder Phase B- (input)
20	Encoder Phase A- (input)
21	Ground
22	Encoder Power (+5VDC or +12VDC, see jumper JP4)
23	Encoder Phase A+ (input)*
24	Encoder Power (+5VDC or +12VDC, see jumper JP4)
25	Encoder Index- (input, active low)
26	Ground

* Use A+ and B+ for single-ended Encoder inputs
** These signals are not suitable for directly driving optically isolated inputs.
*** For use when the computers +12VDC supply is not to be used as the motor drive supply

Mating Connector:26-pin dual-row IDC female, Circuit Assembly P/N 26IDS2-C-SPT-SR or equivalent

DCX MODULE CONNECTOR PIN NUMBERING
(TOP SIDE VIEW)



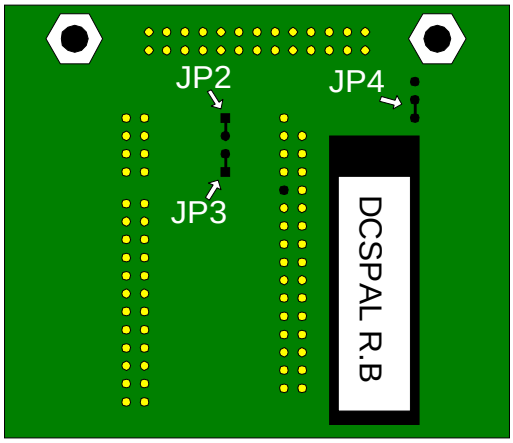
DCX-MC110 Module Configuration Jumpers - configuration in **bold type** denotes default factory shipping configuration

JP2 & JP3 – Encoder type (Single ended versus differential)

Pins	***Description***
1 to 2	**Single ended encoder A, B (pcb trace)**
	Differential encoder A+, A-, B+, B- (cut JP2 and JP3 pcb trace)

JP4 – Encoder Power Select (+5VDC or +12 VDC)

Pins	***Description***
1 to 2	+12 VDC encoder supply on J3 pin 17 (and cut pcb trace from JP4-2 to JP4-3)
2 to 3	**+5 VDC encoder supply on J3 pin 17 (pcb trace)**



JP5 – Encoder Phase Select (consult factory)
JP7 – Voltage / Current Mode Select (consult factory)
JP8 – Current Sense Resistor Defeat (consult factory)
JP9 – Positive Supply Select (consult factory)
JP10 – Negative Supply Select (consult factory)

DCX-MC400 Digital I/O Module

DCX-MC400 Electrical Specifications

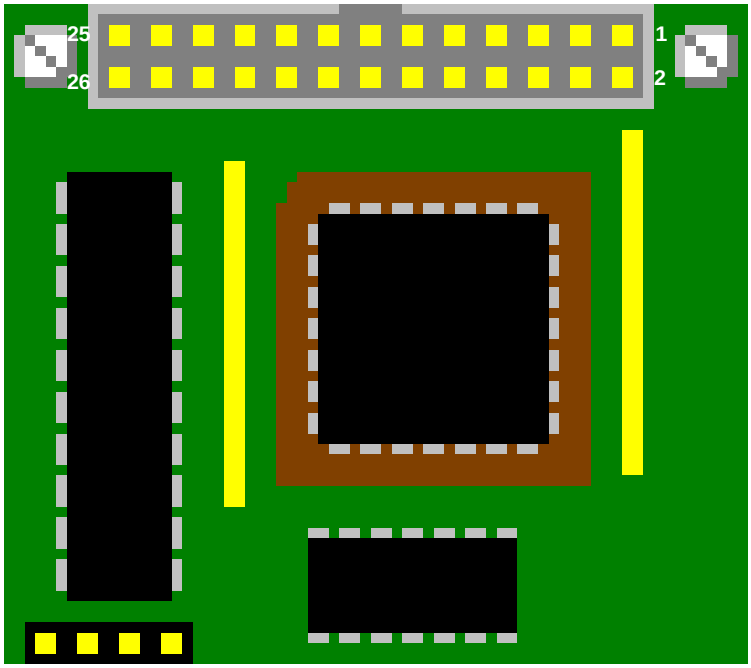
Parameter	*Min.*	*Max*	*Unit*
Digital Input – High voltage	2.0	5.3	V
Digital Input – Low voltage	-0.3	0.8	V
Digital Output – High voltage	2.4		V (current source 0.25ma)
Digital Output – Low voltage		0.4	V (current source 2.0ma)
Input leakage		+/- 10.0	uA

J3 connector pin-out

Pin #	*Description*
1	Digital I/O channel #1
2	Digital I/O channel #2
3	Digital I/O channel #3
4	Digital I/O channel #4
5	Digital I/O channel #5
6	Digital I/O channel #6
7	Digital I/O channel #7
8	Digital I/O channel #8
9	Digital I/O channel #9
10	Digital I/O channel #10
11	Digital I/O channel #11
12	Digital I/O channel #12
13	Digital I/O channel #13
14	Digital I/O channel #14
15	Digital I/O channel #15
16	Digital I/O channel #16
17	Reserved
18	Reserved
19	Reserved
20	+5 VDC
21	Ground
22	Reserved
23	Reserved
24	Reserved
25	Reserved
26	Ground

Mating Connector:26-pin dual-row IDC female, Circuit Assembly P/N 26IDS2-C-SPT-SR or equivalent

DCX-MC400 Module layout



DCX-MC500/510/520 Analog I/O Module

J3 connector pin-out

Pin #	*Description*
1	Channel 1 Input (0 to +5 volts)
2	Channel 1 Output (-10 to +10 volts)
3	Channel 2 Input (0 to +5 volts)
4	Channel 2 Output (-10 to +10 volts)
5	Channel 3 Input (0 to +5 volts)
6	Channel 3 Output (-10 to +10 volts)
7	Channel 4 Input (0 to +5 volts)
8	Channel 4 Output (-10 to +10 volts)
9	Reserved
10	Channel 1 Output (0 to +5 volts)
11	Reserved
12	Channel 2 Output (0 to +5 volts)
13	Reserved
14	Channel 3 Output (0 to +5 volts)
15	Reserved
16	Channel 4 Output (0 to +5 volts)
17	Analog Ground
18	External A/D reference input (see jumper JP1)
19	+12 VDC
20	-12 VDC
21	No connect
22	No connect
23	+5 VDC
24	+5 VDC
25	Digital Ground
26	Digital Ground

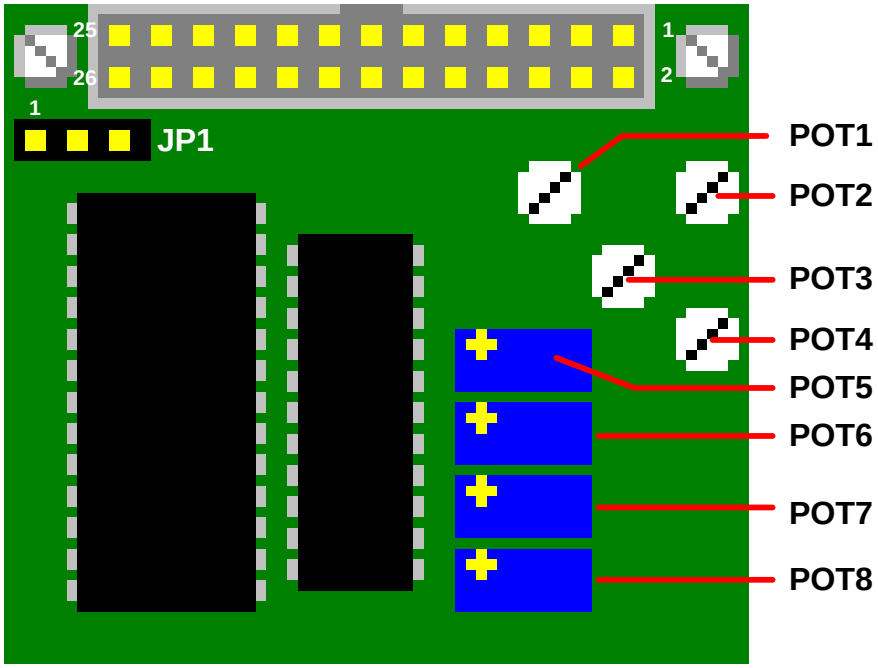
Mating Connector:26-pin dual-row IDC female, Circuit Assembly P/N 26IDS2-C-SPT-SR or equivalent

DCX-MC500/510/520 Module Configuration Jumpers - configuration in **bold type** denotes default factory shipping configuration

JP1 – A/D reference select (external reference or on board +5 VDC reference)

Pins	*Description*
1 to 2	Use external reference (supplied by user on J3 pin 18)
2 to 3	**Use the on board +5 VDC reference**

DCX-MC500 Module layout



DCX-BF022 Relay Rack Interface

J1 connector pin-out - The signals are arranged to interface the DCX-MC400 directly to an OPTO 22 relay rack.

Pin #	Description
1	Digital I/O channel #1
2	Digital I/O channel #2
3	Digital I/O channel #3
4	Digital I/O channel #4
5	Digital I/O channel #5
6	Digital I/O channel #6
7	Digital I/O channel #7
8	Digital I/O channel #8
9	Digital I/O channel #9
10	Digital I/O channel #10
11	Digital I/O channel #11
12	Digital I/O channel #12
13	Digital I/O channel #13
14	Digital I/O channel #14
15	Digital I/O channel #15
16	Digital I/O channel #16
17	No connect
18	No connect
19	No connect
20	+5 VDC
21	Ground
22	No connect
23	No connect
24	No connect
25	No connect
26	Ground

Mating Connector:26-pin dual-row IDC female, Circuit Assembly P/N 26IDS2-C-SPT-SR or equivalent

J2 connector pin-out - The signals are arranged to interface the DCX-AT200 General Purpose I/O (connector J3) directly to an OPTO 22 relay rack.

Pin #	*Description*
1	+5 VDC
2	No connect
3	Digital I/O channel #16
4	No connect
5	Digital I/O channel #15
6	Digital I/O channel #14
7	Digital I/O channel #13
8	Digital I/O channel #12
9	Digital I/O channel #11
10	Digital I/O channel #10
11	Digital I/O channel #9
12	Digital I/O channel #8
13	Digital I/O channel #7
14	Digital I/O channel #6
15	Digital I/O channel #5
16	Digital I/O channel #4
17	Digital I/O channel #3
18	Digital I/O channel #2
19	Digital I/O channel #1
20	No connect
21	No connect
22	No connect
23	No connect
24	Ground
25	No connect
26	Ground

Mating Connector:26-pin dual-row IDC female, Circuit Assembly P/N 26IDS2-C-SPT-SR or equivalent

DCX-BF022 Configuration Jumpers - configuration in **bold type** denotes default factory shipping configuration

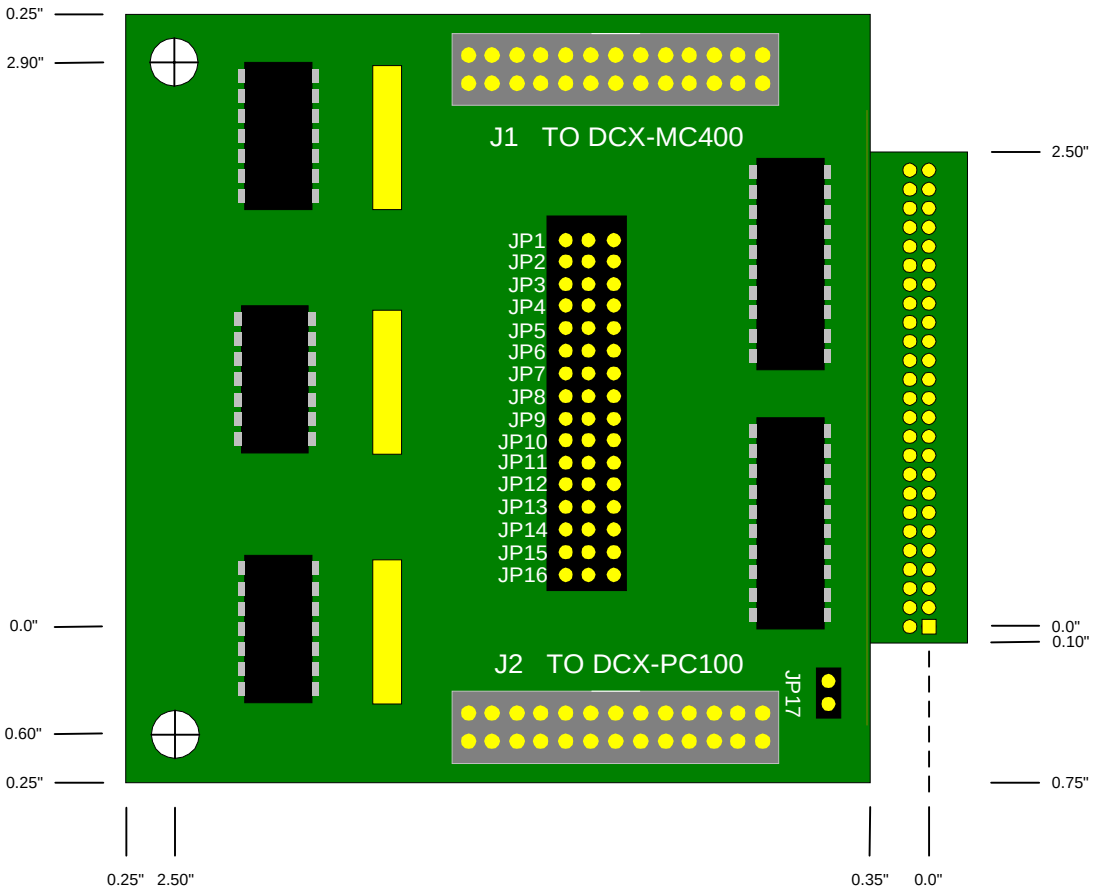
JP1 – JP16 Configure Digital channel as Input or Output

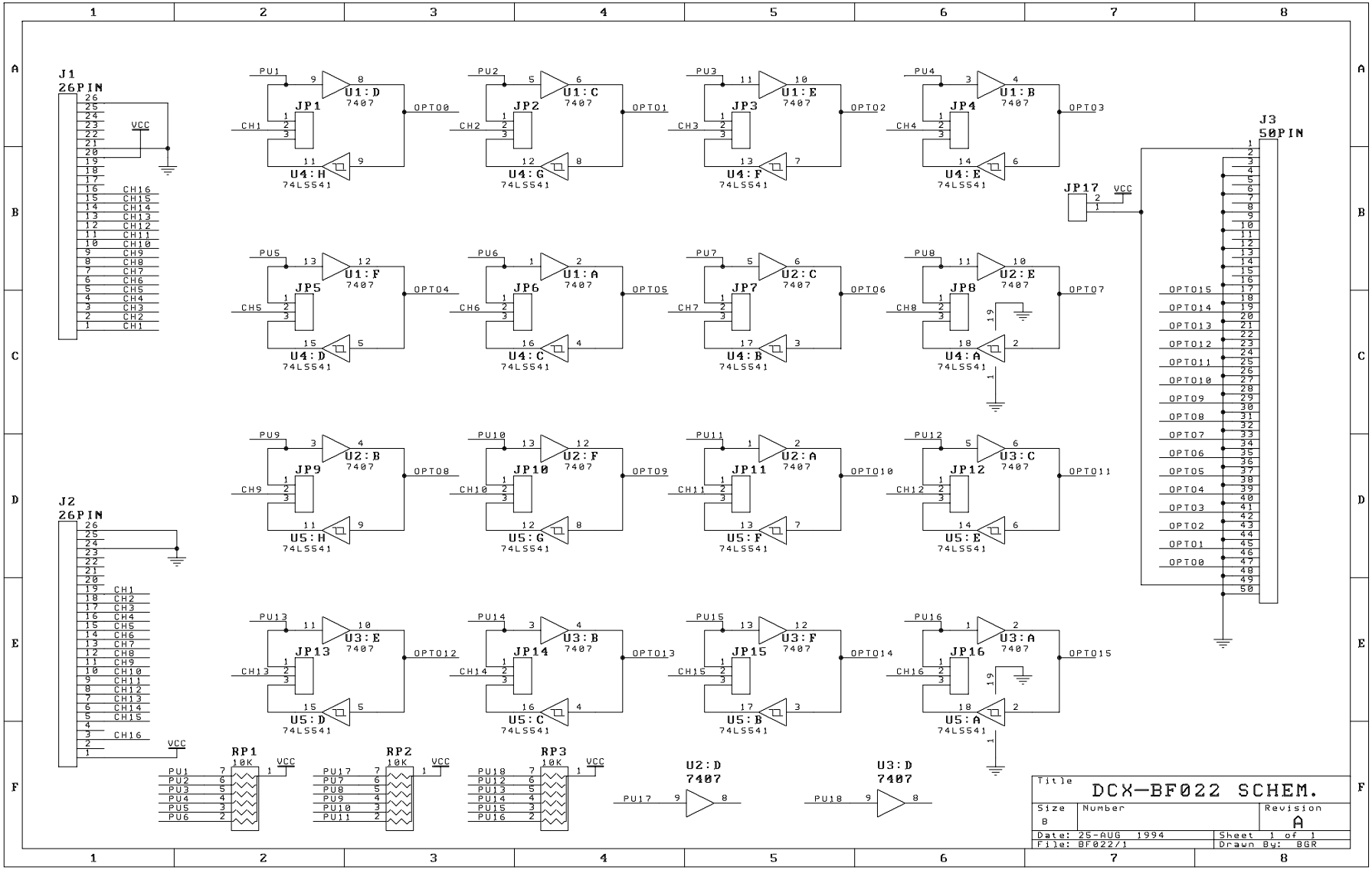
Pins	*Description*
1 to 2	**Configure channel as Output**
2 to 3	Configure channel as an Input

JP17 – Select Relay Rack supply source

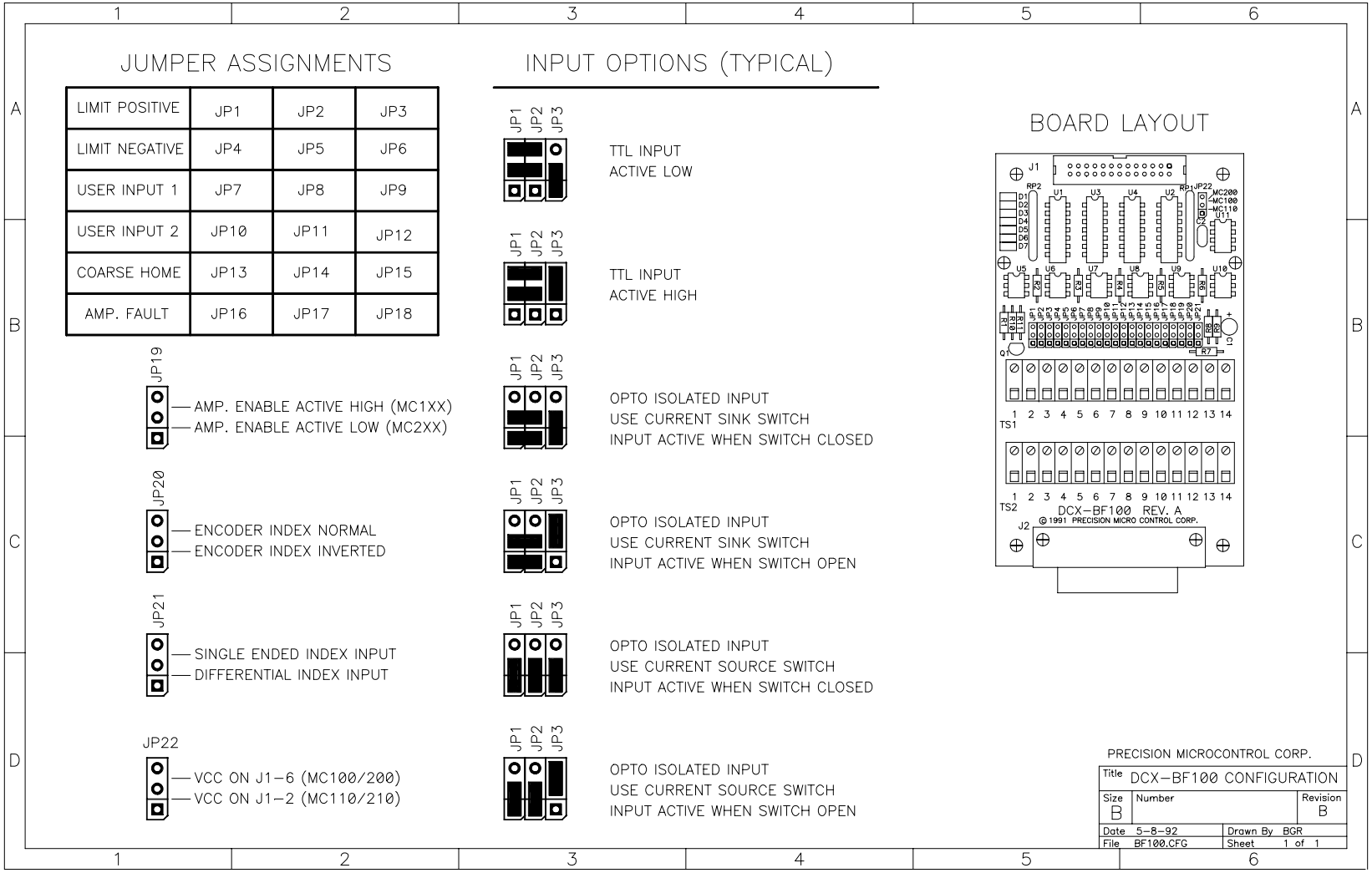
Pins	*Description*
1 to 2	**DCX provides +5 VDC Relay Rack supply**
2 to 3	Relay Rack has separate +5 VDC supply

DCX-BF022 Interface layout




J1
26PIN
J2
26PIN
J3
50PIN
JP17
VCC
U1:D 7407
U1:C 7407
U1:E 7407
U1:B 7407
U1:F 7407
U1:A 7407
U2:C 7407
U2:E 7407
U2:B 7407
U2:F 7407
U2:A 7407
U3:C 7407
U3:E 7407
U3:B 7407
U3:F 7407
U3:A 7407
U4:H 74LS541
U4:G 74LS541
U4:F 74LS541
U4:E 74LS541
U4:D 74LS541
U4:C 74LS541
U4:B 74LS541
U4:A 74LS541
U5:H 74LS541
U5:G 74LS541
U5:F 74LS541
U5:E 74LS541
U5:D 74LS541
U5:C 74LS541
U5:B 74LS541
U5:A 74LS541
RP1 10K
RP2 10K
RP3 10K
U2:D 7407
U3:D 7407
Title
DCX-BF022 SCHEM.
Size B
Number
Revision A
Date: 25-AUG 1994
Sheet 1 of 1
File: BF02271
Drawn By: BGR

DCX-BF100 Servo Module Interconnect Board



DCX-BF100 to DCX-MC100 Connections:

Connector J1

Pin	*Description*
1	Analog Ground
2	Analog Command output
3	+12 VDC
4	-12 VDC
5	Ground
6	+5 VDC
7	No connect
8	No connect
9	Coarse Home
10	Amplifier Fault
11	Amplifier Inhibit
12	No connect
13	No connect
14	Limit +
15	Limit -
16	Encoder Phase B+
17	Encoder Power
18	Ground
19	Encoder Phase A-
20	Encoder Phase B-
21	Ground
22	Encoder Power
23	Encoder Phase A+
24	Encoder Power
25	Encoder Index-
26	Ground

Connector J2

Pin	*Description*
1	Analog Ground
2	+12 VDC
3	Ground
4	Opto Supply
5	Coarse Home
6	Amplifier Enable
7	No connect
8	Limit -
9	Encoder Power
10	Encoder Phase A-
11	Ground
12	Encoder Phase A+
13	Encoder Index-
14	Analog Command output
15	-12 VDC
16	+5 VDC
17	Encoder Index+
18	Amplifier Fault
19	No connect
20	Limit +
21	Encoder Phase B+
22	Ground
23	Encoder Phase B-
24	Encoder Power
25	Encoder Power

Terminal strip TS1

Pin	*Description*
1	Shield
2	Analog Ground
3	Analog Command output
4	+5 VDC
5	+5 VDC
6	Amplifier Enable
7	Coarse Home
8	Amplifier Fault
9	No connect
10	No connect
11	Limit +
12	Limit -
13	Opto Supply
14	Ground

Terminal strip TS2

Pin	*Description*
1	Shield
2	Encoder Power
3	Encoder Phase B+
4	Encoder Phase A-
5	Encoder Phase A+
6	Encoder Phase B-
7	Encoder Index+
8	Encoder Index-
9	Ground
10	Encoder Power
11	Encoder Power
12	Ground
13	+5 VDC
14	Ground

DCX-BF100 to DCX-MC110 Connections:

Connector J1

Pin	Description
1	Motor Drive +
2	Encoder Power
3	Encoder Phase A+
4	Encoder Phase B+
5	Ground
6	Motor Drive -
7	Positive Supply
8	Negative Supply
9	Coarse Home
10	Amplifier Fault
11	Amplifier Inhibit
12	Reserved
13	Reserved
14	Limit +
15	Limit -
16	Encoder Phase B+
17	Encoder Power
18	Ground
19	Encoder Phase B-
20	Encoder Phase A-
21	Ground
22	Encoder Power
23	Encoder Phase A+
24	Encoder Power
25	Prim. Encoder Index-
26	Ground

Connector J2

Pin	Description
1	Motor Drive +
2	Encoder Phase A+
3	Ground
4	Opto Supply
5	Coarse Home
6	Amplifier Enable
7	No connect
8	Limit -
9	Encoder Power
10	Encoder Phase B-
11	Ground
12	Encoder Phase A+
13	Prim. Encoder Index-
14	Encoder Power
15	Encoder Phase B+
16	Motor Drive -
17	Encoder Index+
18	Amplifier Fault
19	No connect
20	Limit +
21	Encoder Phase B+
22	Ground
23	Encoder Phase A-
24	Encoder Power
25	Encoder Power

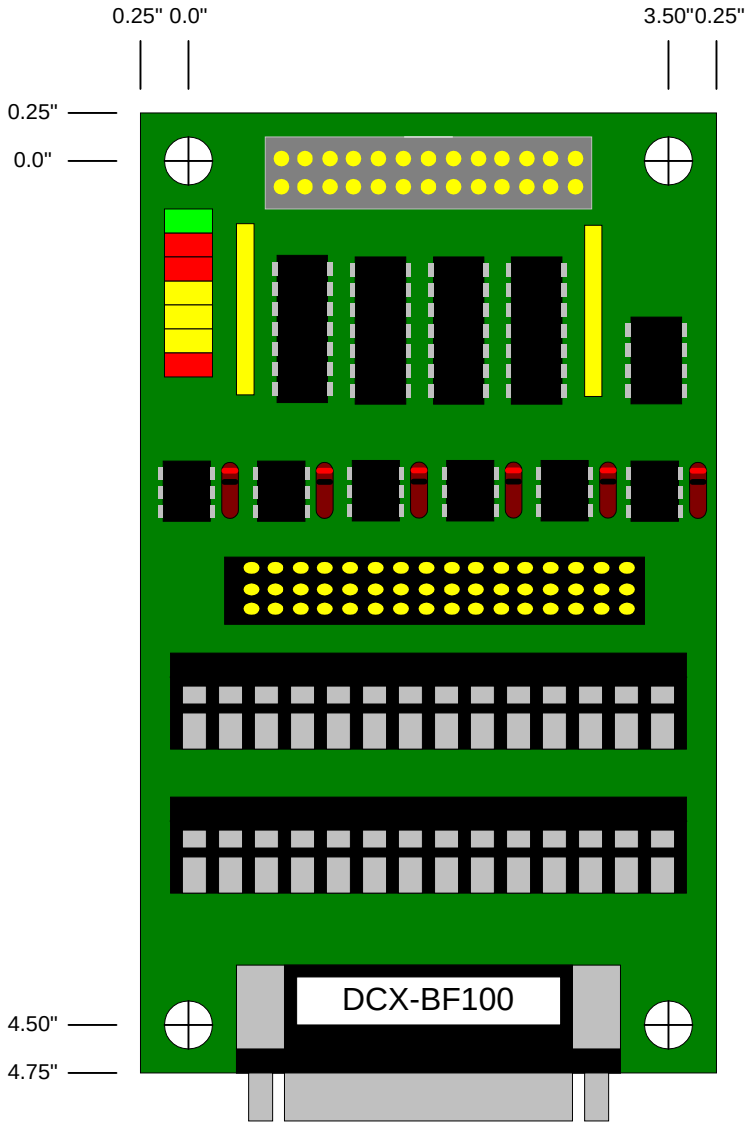
Terminal strip TS1

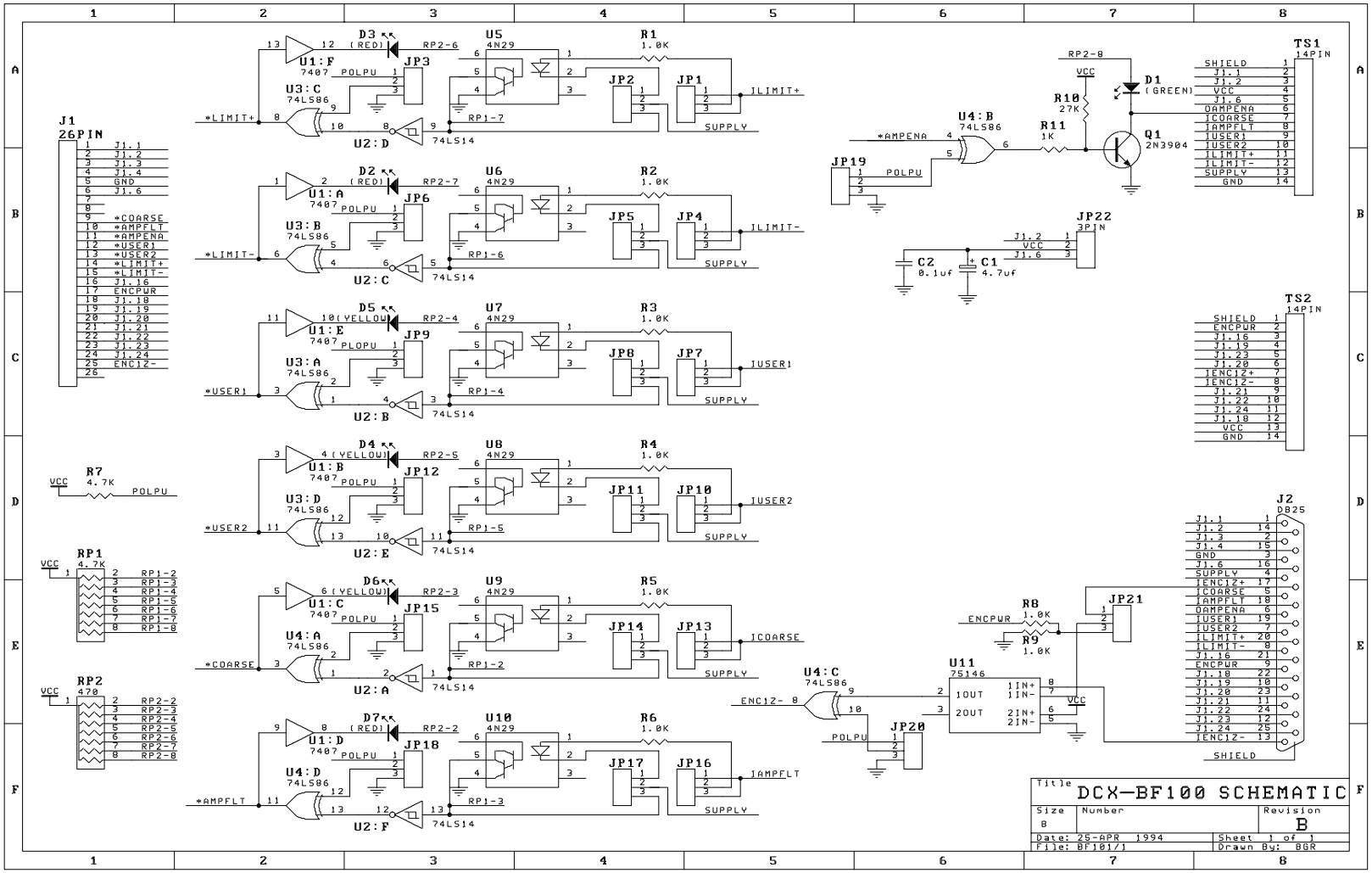
Pin	Description
1	Shield
2	Motor Drive +
3	Encoder Power
4	+5 VDC
5	Motor Drive -
6	Amplifier Enable
7	Coarse Home
8	Amplifier Fault
9	No connect
10	No connect
11	Limit +
12	Limit -
13	Opto Supply
14	Ground

Terminal strip TS2

Pin	Description
1	Shield
2	Encoder Power
3	Encoder Phase B+
4	Encoder Phase B-
5	Encoder Phase A+
6	Encoder Phase A-
7	Encoder Index+
8	Encoder Index-
9	Ground
10	Encoder Power
11	Encoder Power
12	Ground
13	+5 VDC
14	Ground

DCX-BF100 Interface layout




DCX-BF100 SCHEMATIC
Title
Size B
Number
Revision B
Date: 25-APR 1994
Sheet 1 of 1
File: BF101/1
Drawn By: BGR
J1 26PIN
TS1 14PIN
TS2 14PIN
J2 DB25
U1:F 7407
U1:A 7407
U1:E 7407
U1:B 7407
U1:C 7407
U1:D 7407
U2:D 74LS14
U2:C 74LS14
U2:B 74LS14
U2:E 74LS14
U2:A 74LS14
U2:F 74LS14
U3:C 74LS86
U3:B 74LS86
U3:A 74LS86
U3:D 74LS86
U4:B 74LS86
U4:C 74LS86
U4:D 74LS86
U5 4N29
U6 4N29
U7 4N29
U8 4N29
U9 4N29
U10 4N29
U11 75146
D1 (GREEN)
D2 (RED)
D3 (RED)
D4 (YELLOW)
D5 (YELLOW)
D6 (YELLOW)
D7 (RED)
Q1 2N3904
R1 1.0K
R2 1.0K
R3 1.0K
R4 1.0K
R5 1.0K
R6 1.0K
R7 4.7K
R8 1.0K
R9 1.0K
R10 27K
R11 1K
RP1 4.7K
RP2 470
C1 4.7uf
C2 0.1uf
JP1
JP2
JP3
JP4
JP5
JP6
JP7
JP8
JP9
JP10
JP11
JP12
JP13
JP14
JP15
JP16
JP17
JP18
JP19
JP20
JP21
JP22 3PIN
SUPPLY
POLPU
VCC
GND
*LIMIT+
*LIMIT-
*USER1
*USER2
*COARSE
*AMPFLT
*AMPENA
ILIMIT+
ILIMIT-
IUSER1
IUSER2
ICOARSE
IAMPFLT
ENC1Z-
ENCPWR
SHIELD

Chapter Contents

Chapter

26

Command Set Introduction

Introduction to MCCL (low level command set)

The low level platform of all DCX operations is the DCX command set named **MCCL** (**M**otion **C**ontrol **C**ommand **L**anguage). These board level commands are equivalent to the instruction set of a micro controller. These low level commands provide the user access to all DCX operations.

All DCX MCCL commands are made up of two character mnemonic. The characters that make the mnemonic are selected from the command description so that the command has a direct correlation to the operation to be performed. For example, the MCCL command that is used to move an axis to an absolute position is:

MA (**M**ove **A**bsolute).

Any MCCL command that references an axis is preceded by an axis specifier *a* (*a*MA). To issue a move absolute to axis #1:

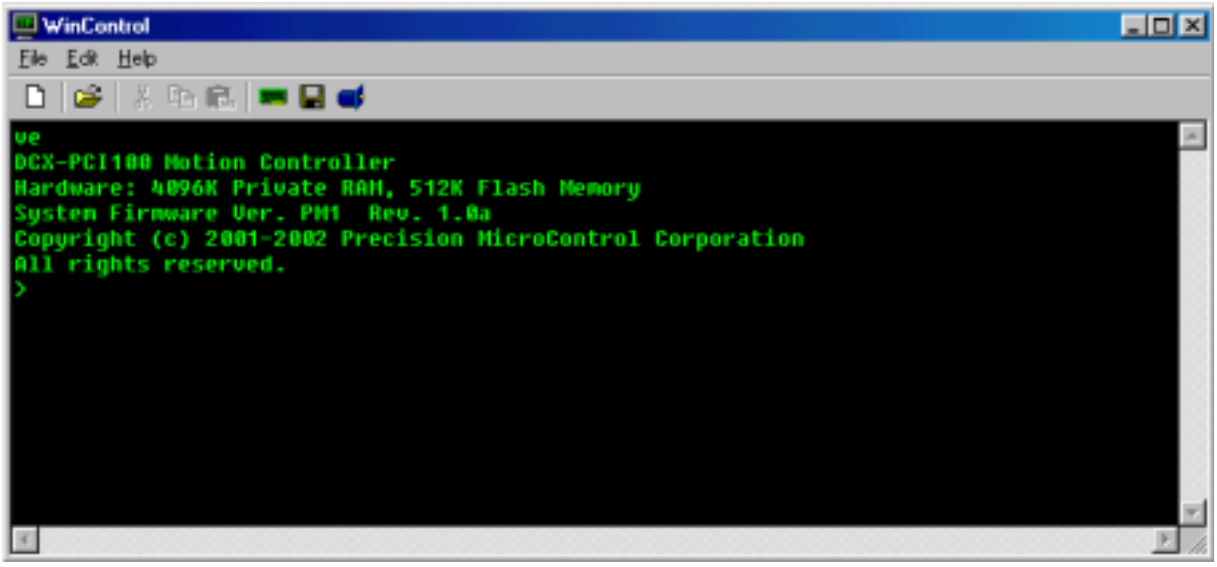
1**MA** (axis #1 **M**ove **A**bsolute)

Most DCX commands will also include a parameter value following the two character mnemonic. This parameter is identified as *n* (*a*MA*n*). To move axis #1 to absolute position 10.25:

1**MA**10.25 (axis #1 **M**ove **A**bsolute to position 10.25)

Included with PMC's MCAPI is the Windows based MCCL command interface utility **WinControl**. This utility allows the user to communicate directly with the DCX in its native language. Any characters

typed by the user on the keyboard will be passed to the DCX input character buffer. The WinControl file menu allows the user to download MCCL text files.



A typical MCCL command description is shown below:

Move Absolute
MCCL command: *a*MR*n* *a* = Axis number *n* = integer or real >= 0
compatibility: MC100, MC110
see also: MR, PM
This command generates a motion to an absolute position n. A motor number must be specified and that motor must be in the 'on' state for any motion to occur. If the motor is in the off state, only its internal target position will be changed. See the description of **Point to Point Motion** in the **Motion Control** chapter.

The ***MCCL command*** line shown the command syntax and parameter type and/or range

The ***compatibility*** line list the DCX modules that support the command

The ***see also*** line list associated MCCL commands

MCCL Command Quick Reference Tables

Setup Commands

MCCL	Code	Description
DH	23h	Define Home
DI	24h	DIrection
FF	33h	amplifier Fault input ofF
FN	32h	amplifier Fault input oN
FR	27h	set derivative sampling period
HL	D3h	set motion High Limit
IL	28h	set Integration Limit
LF	36h	motion Limits ofF
LL	D2h	set motion Low Limit
LM	34h	Limit Mode
LN	35h	motion Limits oN
SA	2Bh	Set Acceleration
SD	2Ch	Set Derivative gain
SE	19h	Stop on Error
SG	2Dh	Set prop. Gain of motor
SI	2Eh	Set Integral gain
SV	2Fh	Set Velocity
UA	9Ch	Use as default Axis
UK	D7h	set User output constant
UO	B3h	set User Offset
UP	9Dh	Use Physical axis
UR	B1h	set User Rate conversion
US	AFh	set User Scale
UT	B2h	set User Time conversion
UZ	B0h	set User Zero

Motion Commands

MCCL	Code	Description
AB	Ah	ABort
FE	Bh	Find Edge
FI	Ch	Find Index
GH	Dh	Go Home
GO	Eh	GO
HO	Fh	HOme
LP	70h	Learn Position
LT	71h	Learn Target
MA	10h	Move Absolute
MF	11h	Motor ofF
MN	13h	Motor oN
MP	14h	Move to Point
MR	15h	Move to Point
PR		Record motion data
ST	16h	STop

Mode Commands

MCCL	Code	Description
PM	17h	enable Position Mode
VM	18h	enable Velocity Mode

Reporting Commands

MCCL	Code	Description
DO		Display recorded optimal position
DR		Display recorded actual position
TA	49h	Tell Analog to digital converter
TB	5Bh	Tell Breakpoint position
TC	4Ah	Tell Channel
TD	4Bh	Tell Derivative gain
TE		Tell command interface Error
TF	4Dh	Tell Following error
TG	4Eh	Tell proportional Gain
TI	4Fh	Tell Integral gain
TL	50h	Tell integration Limit
TM	51h	Tell stored Macros
TO	59h	Tell Optimal
TP	52h	Tell Position
TR	57h	Tell Register n
TS	53h	Tell Status
TT	54h	Tell Target
TV	55h	Tell Velocity
TZ	5Ah	Tell index position
VE	56h	tell VErsion

Macro Commands

MCCL	Code	Description
BK	79h	BreaK
ET	FBh	Escape Task
GT	FAh	Generate Task
MC	2h	Macro Call
MD	3h	Macro Definition
MJ	5h	Macro Jump
RM	4h	Reset Macros
TM	51h	Tell Macros

I/O Commands

MCCL	Code	Description
CF	1Fh	Channel ofF
CH	42h	Channel High true logic
CI	20h	Channel In
CL	43h	Channel Low true logic
CN	21h	Channel oN
CT	22h	Channel ouT
GA		Get Analog
OA		Output Analog
TA	49h	Tell the value of Analog input
TC	4Ah	Tell state of digital Channel
WF	67	Wait for channel ofF
WN	68	Wait for channel oN

Register Commands

MCCL	Code	Description
AA	85h	Accumulator Add
AC	8Ch	Accumulator Complement
AD	88h	Accumulator Divide
AE	8Fh	Accumulator logical Exclusive or
AL	82h	Accumulator Load
AM	87h	Accumulator Multiply
AN	8Dh	Accumulator logical aNd with n,
AO	83h	Accumulator logical Or with n
AR	84h	copy Accumulator to Register n
AS	86h	Accumulator Subtract
AV	8Bh	Accumulator eValuate
AX	E1h	get Aux. indeX position
GA	F8h	Get Analog value
GD		Get module ID
GU	89h	Get the default axis
LU	81h	Look Up motor table variable
OA	F9h	Output Analog value
RA	83h	copy Register to Accumulator
RB	96h	Read Byte into accumulator
RD	93h	Read Double into accumulator
RL	98h	Read Long into accumulator
RV	92h	Read float into accumulator
RW	97h	Read Word into accumulator
SL	90h	Shift Left accumulator n bits
SR	91h	Shift Right accumulator n bits
TR	57h	Tell contents of Register n

Sequence Commands

MCCL	Code	Description
DF	6B	Do if channel ofF
DN	6A	Do if channel oN
IB	A5	If Below do next command
IC	A1	If Clear, do next command
IE	A2	If Equals do next command
IF	6D	If channel ofF do next command
IG	A4	If accumulator is Greater do next
IN	6C	If channel oN do next command
IP	60	Interrupt on absolute Position
IR	61	Interrupt on Relative position
IS	A0	If bit Set do next command
IU	A3	If Unequal do next command
JP	6	JumP to command absolute
JR	7	Jump to command Relative
RP	64	RePeat
WA	65	WAit (time)
WE	66	Wait for Edge
WF	67	Wait for channel ofF
WI	5E	Wait for Index
WN	68	Wait for channel oN
WP	62	Wait for absolute Position
WR	63	Wait for Relative position
WS	63	Wait for Stop
WT	C6	Wait for Target

Miscellaneous Commands

MCCL	Code	Description
DM	3Ch	Decimal Mode
DW	FDh	Disable Watchdog
FD		Format text with Doubles
FT		Format Text with Integers
HM	3Dh	Hexadecimal Mode
NO	78h	No Operation
OD		Output text with Doubles
OT		Output Text with integers
PC	80h	set Prompt Character
RT	2Ah	ReseT system

Building MCCL Macro Sequences

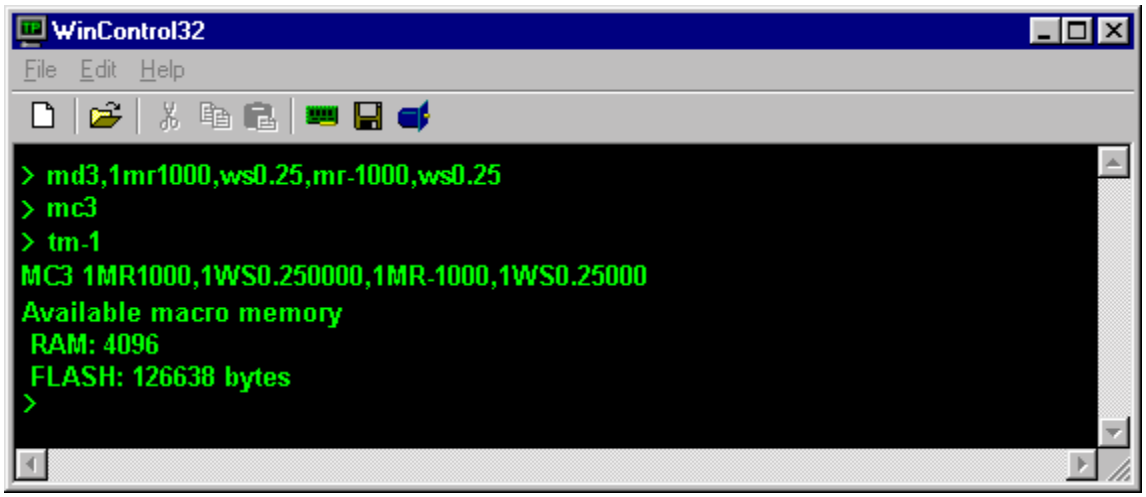
A powerful feature of the DCX is the ability to define MCCL (Motion Control Command Language) command sequences as macros. This simply means defining a mnemonic that will execute a user defined sequence of commands. For example:

```
1MR1000,WS0.25,MR-1000,WS0.25
```

will cause the motor attached to axis 1 to move 1000 counts in the positive direction, wait one quarter second after it has reached the destination, then move back to the original position followed by a similar delay. If this sequence were to represent a frequently desired motion for the system, it could be defined as a macro command. This is done by inserting a Macro Define (MD) command as the first command in the command string. For example:

```
MD3,1MR1000,WS0.25,MR-1000,WS0.25
```

will define macro #3. Whenever it is desired to perform this motion sequence, issue the command Macro Call (MC3). To command the DCX to display the contents of a macro, issue the **T**ell **M**acro (**TM*n***) command with parameter 'n' = the number of the macro to be displayed. To display the contents of all stored macro's issue the Tell macro command with parameter 'n' = -1.



Once a macro operation has begun, the host will not be able to communicate with the DCX until the **macro has terminated**. For information on communicating with the controller while executing macro's please refer to the section titled **MCCL Multi-Tasking**.

The DCX can store up to 1000 user defined macros. Each macro can include as many as 255 bytes. Depending on the type of command and type of parameter, a command can range from 2 bytes (a command with no parameter) to 10 bytes (a command with a 64 bit floating point parameter).

All memory on the DCX-PCI100 is volatile, which means that the data in memory will be cleared when the controller is reset or power to the board is turned off. The **R**eset **M**acro (**RM*n***) command is used to erase macros.

Since the DCX provides no protection against overflowing the macro storage space, it is suggested that the user monitor the amount of memory available for macro storage. The **T**ell **M**acro (**TM*n***) command can be used to display the amount of RAM memory available for macros storage at any give time.

To terminate the execution of any macro that was started from WinControl press the escape key. To start a macro that runs indefinitely without 'locking up' communication with the host, start the macro's with the ***generate a Background task*** (GT) command instead of the ***Call macro command*** (MC). This will allow the operations called by macro 0 to execute as a background task. Please refer to the next section **Multi-Tasking**.

MCCL Multi-Tasking

The DCX command interpreter is designed to accept commands from the user and execute them immediately. With the addition of sequencing commands, the user is able to create sophisticated command sequences that run continuously, performing repetitive monitoring and control tasks. The drawback of running a continuous command sequence is that the command interpreter is not able to accept other commands from the user.

Once a macro operation has begun, the host will not be able to communicate with the DCX until the **macro has terminated**.

The DCX supports Multi-tasking, which allows the controller to execute continuous monitoring or control sequences as background tasks while the foreground task communicates with the 'host'.

With the exception of ***reporting commands*** (Tell Position, Tell Status, etc...), any MCCL commands can be executed in a background task. Prior to executing a command sequence/macro as a background task, the ***user should always test the macro by first executing it as a foreground task***. When the user is satisfied with the operation of the macro, it can be run as a background task by issuing the **G**enerate **T**ask (**GT*n***) command, specifying the macro number as the command parameter. After the execution of the Generate Task command, the accumulator (register 0) will contain an identifier for the background task. Within a few milliseconds, the DCX will begin running the macro as a background task in parallel with the foreground command interpreter. The DCX will be free to accept new commands from the user.

```
;Multitasking example – while axis #1 is moving, monitor the state of digital
;input #4. When the input goes active, stop axis #1 and terminate the
;background task


AL0,AR10                          ;define user register 10 as input #4 active
                                  ;flag register
AL0,AR100                         ;define user register #100 as background task
                                  ;ID register

MD100,IN4,MJ101,NO,1JR-3          ;jump to macro 101 when digital input #4
                                  ;turns on
MD101,1ST,1WS.05,AL1,AR10,ET@100  ;stop axis #1. Terminate background task

GT100,AR@100,1VM,1DI0,1GO         ;spawn macro #10 as background task. Store
                                  ;task ID into register #100. Start axis #1
                                  ;moving in velocity mode,
```

Note: Immediately after 'spawning' the background task (with the GTn command), the value in the accumulator (task identifier) should be stored in a user register. This value will be required to terminate execution of the background task.

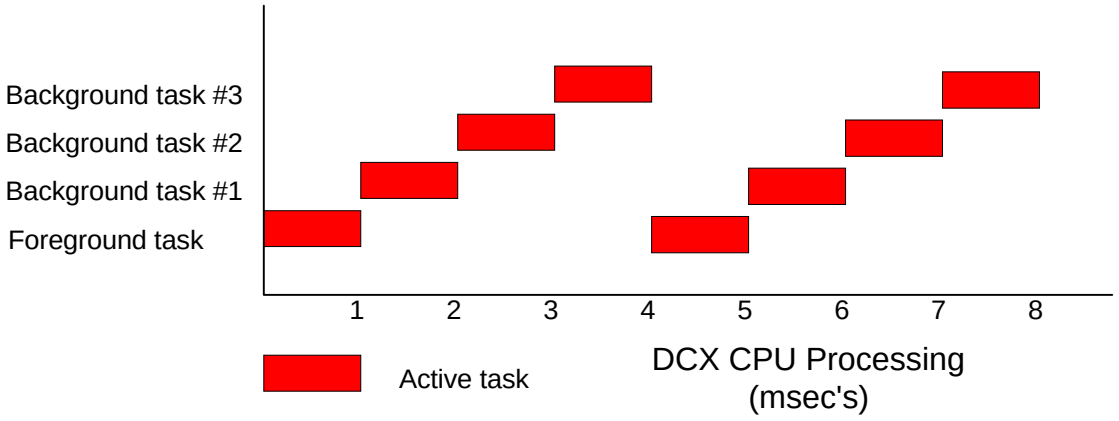
Another way to create a background task is to place the Generate Task command as the first command in a command line, using a parameter of 0. This instructs the command interpreter to take all the commands that follow the Generate Task command and cause them to run as a background task. The commands will run identically to commands placed in a macro and generated as a task.

```
;Multitasking example – while axis #1 is moving, monitor the state of the
;motor error status bit (bit 7). If error occurs set bit #1 of user
;register 200

GT0,AR@100,LU"STATUS",1RL@0,IC7,JR-3,NO,AL1,AR200,ET@100
                                    ;loop on axis #1 status bit 7, if set; set
                                    ;bit #1 of register 200, terminate task using
                                    ;Task ID (in register #100)
```

Within the background task, the commands can move motors, wait for events, or perform operations on the registers, totally independent of any commands issued in the foreground. However, the user must be careful that they do not conflict with each other. For example, if a background task issues a move command to cause a motor to move to absolute position +1000, and the user issues a command at the same time to move the motor to -1000, it is unpredictable whether the motor will go to plus or minus 1000.

In order to prevent conflicts over the registers, the background task has its own set of registers 0 through 9 (register 0 is the accumulator). These are private to the background task and are referred to as its 'local' registers. The balance of the registers, 10 through 255, are shared by the background task and foreground command interpreter, they are referred to as 'global' registers. If the user wishes to pass information to or from the background task, this can be done by placing values in the global register. Note that when a task is created, an identifier for the task is stored in register 0 of both the parent and child tasks.

The DCX is able to run multiple background tasks, each with their own set of registers, but can only have one foreground command interpreter. The maximum number of background tasks is 10. Each background task and the foreground command interpreter get an equal share of the DCX processor's time. When one or more background tasks are active the DCX Task Handler will begin issuing local DCX interrupts every 1 milisecond. Each time the task handler interrupt is asserted, the DCX will switch from executing one task to the next. For example if three background tasks are active, plus the foreground task (always active), each of the four tasks will receive 1 msec of processor time every 4 msec's.



While a background task executes a **Wait** command, that task no longer receives any processor time. For tasks that perform monitoring functions in an endless loop, the command throughput of the DCX can be improved by executing a **Wait** command at the end of the loop until the task needs to run again.

A common way for a background task to be terminated, is when the command sequence of the task finishes execution. This will occur at the end of the macro or if a **B**rea**K** (**BK**) command is executed. When a task is terminated, the resources it required are made available to run other background tasks.

```
;Multitasking example – this background task will terminate itself if the
;motor error status bit for axis #1 is set. This sequence is similar to the
;previous example except that the task is self terminating, so register #100
is not required.

GT0,LU"STATUS",1RL@0,IC7,JR-3,NO,AL1,AR200
                                ;loop on axis #1 status bit 7, if set; set
                                ;bit #1 of register 200, task self terminates
                                ;(no commands left to execute)
```

Alternatively, the **E**scape **T**ask (**ET*n***) command can be used to force a background task to terminate. When a task is generated by the **GT** command, a value known as the **Task ID** is placed into the accumulator. This value should immediately be copied into a user register. The parameter to this command must be the value that was placed in accumulator (register 0) of the parent task, when the Generate Task command was issued.

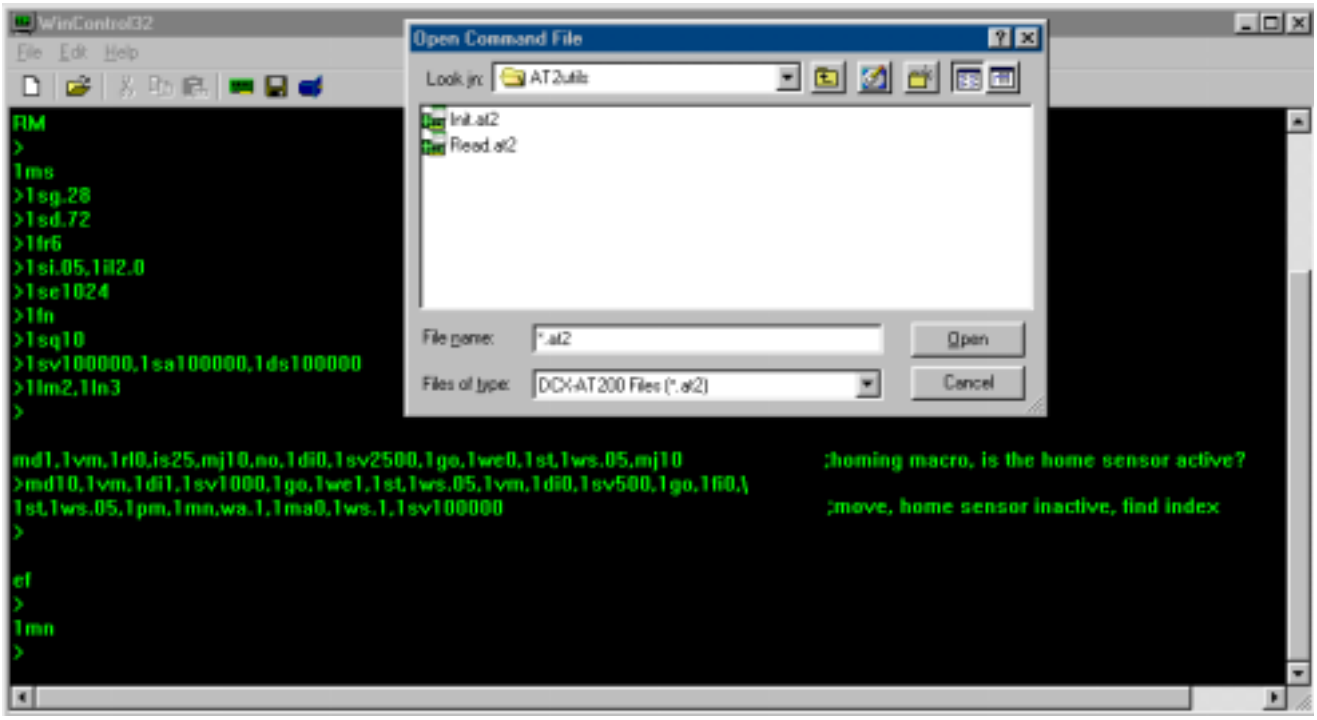
```
;Multitasking example – Terminating a background task with the Escape Task
command.

GT100,AR@150                    ;call macro #100 as a background task, copy
                                ; task ID into user register 150

ET@150                          ;to terminate background task issue escape
                                ; task command with parameter n = Task ID
```

Downloading MCCL Text Files

Motion Control Command Language (MCCL) command sequences can be downloaded as text files to the DCX-AT200. If these command sequences are not defined as macro's (MDn) then the commands will be executed as they are received by the card. If the command sequences are defined as macro's they will be stored in the memory of the DCX-AT200 for execution at a later time.

While most applications will utilize the high level language (C++, VB, Delphi, LabVIEW, etc..) function calls to program the operation of the machine, downloaded MCCL text files are typically used for initial system integration, defining homing routines, and programming background tasks.

The graphic below is a screen capture of PMC's WinControl . This utility provides the user with a direct interface to the DCX., A MCCL text file (init.at2) containing servo parameters and a homing routine have been downloaded to the DCX using the File – Open menu options.



Note: Any characters that are preceded by a semicolon are treated as documenting commands. These documenting character strings are displayed by WinControl but they are 'stripped' from the file and are not be passed to the DCX.

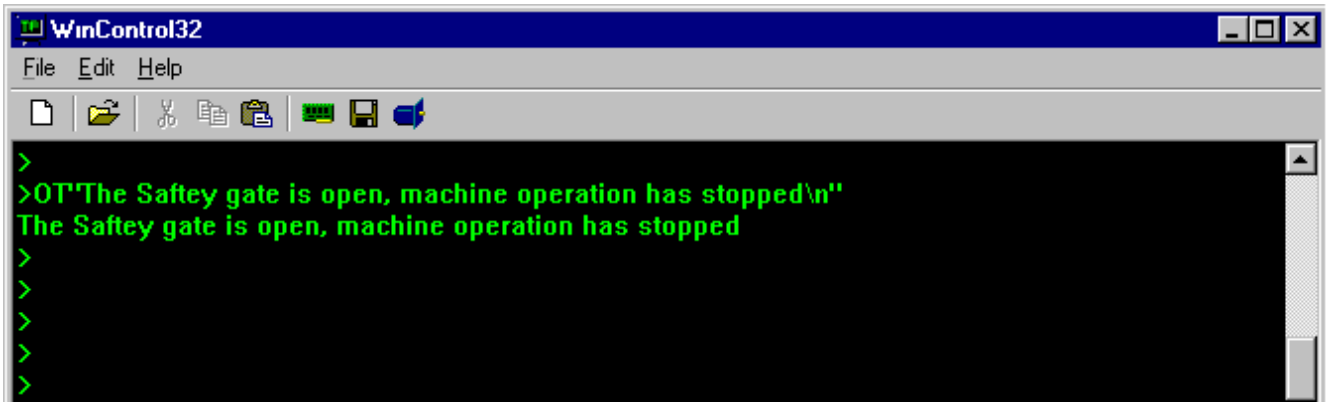
Outputting Formatted Message Strings

The DCX supports the outputting of formatted text strings from the ASCII interface using the Output Text commands. The two commands supported are:

Output Text with integer values (OT" ")
Output text with Double values (OD" ")

The syntax of these two commands are patterned after standard 'C' function 'printf'. For specific 'printf' description please refer to the Microtech Research Inc. MCC960 compiler documentation. The message to be displayed should be delimited by double quotes. Please refer to the examples below:

```
OT"The Saftey gate is open, machine operation has stopped \n"
                                        ;output simple text message,
                                        ; \n = line feed
```



As with typical implementations of 'C' print statements, the DCX supports variables. Prior to executing the output text command, load the accumulator with the data to be included as a variable. In the following example, the Output Double (OD" ") command is used to report the current position of axis one as a floating point value. The % character indicates that a variable stored in the accumulator will be included in the text message. The 'f' indicates that the variable is a floating point value. The '\r' calls for a carriage return at the end of the message.

```
1RD20,OD"The current position of Axis #1 %f \r"
                                        ;load the accumulator with the
                                        ;position of axis #1. Output a text
                                        ;message displaying the position of
                                        ;axis #1 (floating point value),
                                        ;carriage return
```

Reading Data from DCX Memory

A group of read commands are available for accessing the internal Motor Tables of the DCX. These commands provide an easy method of moving motor data in and out of the Accumulator (user register 0).

The **L**ook **U**p (**LU*s***) command is used to load the internal address of a motor table entry into the accumulator. String parameter *s* defines the variable name of the target motor table entry. A Read command (RB, RW, RL, RV, or RD) is then used to load the accumulator with the data from the target motor table entry. The Read command must include the axis specifier 'a'. The type of command to use (byte, double, long, float or word), is determined by the type of data to be accessed and is listed below.

a*RB*n Read Byte (8 bit) at memory location n into accumulator (ACC = (n))
a*RD*n Read Double at memory location n into accumulator (ACC = (n))
a*RL*n Read Long (32 bit) at memory location n into accumulator (ACC = (n))
a*RV*n Read float at memory location n into accumulator (ACC = (n))
a*RW*n Read Word (16 bit) at memory location n into accumulator (ACC = (n))

Examples of using the read commands to access the motor tables are shown below.

To load the 32 bit status of axis 2 into the accumulator, issue the following command sequence:

```
LU"STATUS",2RL@0                ;load the motor table address for axis
                                ;status into the accumulator. Load the
                                ;32 bit status word of axis #2 into the
                                ;accumulator
```

To load the 64 bit current position of axis 3 into the accumulator, issue the following command:

```
LU"POSITION",3RD@0              ;load the motor table address for current
                                ;position into the accumulator. Load the
                                ;accumulator with the 64 bit current
                                 position of  ;axis #3
```

Motor Table Variables – 32 bit integer (long)

Motor Table Variable Description	***Variable Name***
Module Base Address	MODADDR
Motor Status – primary	STATUS
Motor Status - auxiliary	AUXSTAT
Position - Adjustment (index + Offset)	CNTADJ
Position – Current (raw count)	POSCOUNT
Position - Index Count	IDXCOUNT
Position - Optimal (raw count)	CMDCOUNT
Position – Target (raw count)	TGTCOUNT

Motor Table Variables – 64 bit floating point (double)

Motor Table Variable Description	Variable Name
Following Error Setting - maximum	MAXERROR
Position - Current	POSITION
Position – Target	TARGET
Position - Optimal	OPTIMAL
Position - Breakpoint	BRKPOS
Programmed Acceleration	PGMACC
Programmed Velocity	PGMVEL
Scaling - User	SCALE
Scaling - User Offset	OFFSET
Scaling - User Output Constant	OUTCONST
Scaling - User Rate Conversion	RATE
Scaling - User Zero	ZERO
Soft Motion Limit Setting (low)	LOLIM
Soft Motion Limit Setting (high)	HILIM
Velocity - Current	CURVEL

Motor Table Variables – 32 bit floating point (float)

Motor Table Variable Description	Variable Name
PID - Proportional Gain setting	KPOS
PID - Derivative Gain setting	KDER
PID - Integral Gain	KINT
PID - Integration Limit	ILIM

Motor Table Variables – 16 bit integer (word)

Motor Table Entry Description	Offset (decimal)
Axis Number	AXISNUM
Module - Axis	MODAXIS
Module – Base address	MODADDR
Module - Location	MODULE
Module Status	MODST
Module - Type	MODTYPE
Sampling Frequency	SFREQ
Timer - Wait Stop	WAITSTOP
Timer - Wait Target	WAITTARGET

DCX User Registers

The DCX contains 256 general purpose global registers that can be used for; storing command parameters, performing math computations and controlling command execution. The registers are numbered 0 through 255, with register 0 being the 'accumulator'. The accumulator (register 0) is used by all commands that manipulate register data.

Each register can hold a 32 bit integer, a 32 bit single precision floating point number, or a 64 bit double precision floating point number. A register will be loaded with the double precision floating point number if the Accumulator Load (ALn) command is issued with a parameter containing a decimal point. Otherwise, the register will be loaded with a 32 bit integer. When executing commands that perform math operations on the accumulator (AA, AD, AM, ...), the result will have the same precision as the command parameter or the accumulator (prior to the command), whichever is more precise. Since the 32 bit integer is considered to be the least precise, multiplying an integer by a floating point number will always result in a floating point number. If a floating point indirect parameter is used for a command that does not support floating point parameters (eg. CN, LM, PC,...), the register contents will be rounded to the nearest integer prior to use.

Typically the user issues commands with 'immediate' parameters (ie: the parameter ‘n’ is a constant). The user can also issue commands, specifying that the parameter is the contents of a register. This is done by replacing the command parameter with the register number preceded with an '@' sign. For example, the command "1MR@10" will cause the DCX to move axis 1 by the number stored in register 10. The use of a register specifier can be used in any command as the parameter. The DCX ***does not*** support the use of the '@' sign in front of an axis number. The following commands are available for working with the registers:

MCCL Command	*Description*
AAn	Accumulator Add (ACC = ACC + n)
ACn	Accumulator Complement, bit wise (ACC = !ACC)
ADn	Accumulator Divide (ACC = ACC/n)
AEn	Accumulator logical Exclusive or with n, bit wise (ACC = ACC eor n)
ALn	Accumulator Load with constant n (ACC = n)
AMn	Accumulator Multiply(ACC = ACC x n)
ANn	Accumulator logical aNd with n, bit wise (ACC = ACC and n)
AOn	Accumulator logical Or with n, bit wise (ACC = ACC or n)
ARn	copy Accumulator to Register n (REGn = ACCn)
ASn	Accumulator Subtract (ACC = ACC - n)
GAx	Get Analog value (ACC = channel x)
IBn	If accumulator is Below (>) n, do next command, else skip 2 commands
ICn	If bit n of accumulator is Clear, do next command, else skip 2 commands
IEn	If accumulator Equals constant n, do next command, else skip 2 commands
IGn	If accumulator is Greater than ‘n’, do next command, else skip 2 commands
OAx	Output Analog value (channel x = ACC)
ISn	If bit n of accumulator is Set, do next command, else skip 2 commands
IUn	If accumulator is Unequal to ‘n’, do next command, else skip 2 commands
RAn	copy Register n to Accumulator (ACC = REGn)
SLn	Shift Left accumulator n bits (ACC = ACC << n)
SRn	Shift Right accumulator n bits (ACC = ACC >> n)
TRn.p	Tell contents of Register n
TR.p	Tell contents of accumulator (register 0)

Chapter Contents

Chapter

27

MCCL Setup Commands

DH — Define Home

MCCL command: *a*DH*n* *a* = Axis number *n* = integer or real >= 0
compatibility: MC100, MC110
see also: FI, IA, WI

Defines the current position of a motor to be *n*. From then on, all positions reported for that motor will be relative to that point.

DI — DIrection

MCCL command: *a*DI*n* *a* = Axis number *n* = 0, 1
compatibility: MC100, MC110
see also: GO, VM

Sets the move direction of a motor when in velocity mode. A parameter value of 0 results in motion in the positive direction, a value of 1 causes motion in the negative direction.

FF — amplifier Fault oFf

MCCL command: *a*FF *a* = Axis number
compatibility: MC100, MC110
see also: FN

Disables the Amplifier Fault input of a servo control module. See description of amplifier Fault input oN command (FN), for further details.

FN — amplifier Fault oN

MCCL command : aFN*n* *a* = Axis number *n* = integer (0, 1, 2, 128, 129, 130)
compatibility: MC100, MC110
see also: FF

Enables the Amplifier Fault input of a servo control module. If the input goes active after this command is executed, the axis will stop and the amplifier fault tripped flag in servo status will be set. The tripped flag will remain set until the motor is turned back on with the MN command.

Amplifier Fault Mode (aFNn)

Desired action	***Parameter n =***
Turn motor off (disable PID)	0
Stop the motor abruptly (under PID control)	1
Decelerate and stop the motor (under PID control) using the current deceleration setting.	2
Invert the active level of the Amplifier Fault input (add 128 to 0, 1, or 2)	128

FR set the derivative sampling period

MCCL command: *a*FR*n* *a* = Axis number *n* = integer >= 0
compatibility: MC100, MC110
see also: SD, SG

Helps tune servo loop to the inertial characteristics of system. High inertial loads normally require a longer period and low inertial loads a shorter period. The default value is 0 (0.000341 seconds). For a value of n, the derivative sampling period will be (n +1) * sample period (0.000341). See **Tuning the Servo** section in the **Motion Control** chapter.

HL **H**igh motion soft **L**imit

MCCL command: *a*HL*n* *a* = Axis number *n* = integer or real
compatibility: MC100, MC110
see also: LF, LL, LM, LN

This command sets the high limit for motion. After this command is issued, and the motion limit is enabled with the Limit oN (aLNn) command, the command parameter is used as a 'soft' limit for all motion of the axis. If the desired or true position of the axis is greater than this limit, and the axis is being commanded to move in the positive direction, the Soft Motion Limit High and the Motor Error flags in the motor status will be set. The axis will also be turned off, stopped abruptly, or stopped smoothly, depending upon the mode set by the Limit Mode command. Please refer to the **Motion Limits** description in the **Motion Control** chapter.

IL **I**ntegration **L**imit

MCCL command: *a*IL*n* *a* = Axis number *n* = integer >= 0
compatibility: MC100, MC110
see also: SI, SG

Limits level of power that integral gain can use to reduce the position error. The default units for the command parameter are (encoder counts) * (sample interval). See the description of **Tuning the Servo** section in the **Motion Control** chapter.

LF motion **L**imits o**F**f

MCCL command: aLF*n* *a* = Axis number *n* = (see Limit oN table)
compatibility: MC100, MC110
see also: LN, LM

Disables one or more 'hard' limit switch inputs or 'soft' position limits for an axis. The parameter to this command determines which limits will be disabled. The coding of the parameter is the same as for the motion Limits oN command (LN). See the description on **Motion Limits** in the **Motion Control** chapter.

LL **L**ow motion soft **L**imit

MCCL command: aLL*n* *a* = Axis number *n* = integer or real
compatibility: MC100, MC110
see also: LF, LH, LM, LN

This command sets the low limit for motion. After this command is issued, and the motion limit is enabled with the Limit oN (aLNn) command, the command parameter is used as a 'soft' limit for all motion of the axis. If the desired or true position of the axis is less than this limit, and the axis is being commanded to move in the negative direction, the Soft Motion Limit Low and the Motor Error flags in the motor status will be set. The axis will also be turned off, stopped abruptly, or stopped smoothly, depending upon the mode set by the Limit Mode command. See the description of **Motion Limits** in the **Motion Control** chapter.

LM **L**imit **M**ode

MCCL command: aLM*n* *a* = Axis number *n* = integer (see table below)
compatibility: MC100, MC110
see also: LF, LN

This command is used to select how the DCX will react when a 'hard' limit switch or a 'soft' position limit is tripped by an axis. The command parameter should be formed by adding a value of 1, 2, or 3 for the hard limit switch mode, to a value of 4, 8, or 12 for the soft position limit mode. In all cases the Motor Error and one of Limit Tripped flags in the status word will be set when a limit event occurs. This will prevent the DCX from moving the motor until a Motor oN command is issued. See the description of **Motion Limits** in the **Motion Control** chapter.

Limit Mode (aLMn) command

Desired action	***Parameter n =***
Turn motor off (disable PID) when **hard limit sensor 'goes' active** or soft motion limit is exceeded	0,0 *
Stop the motor abruptly (under PID control) when **hard limit sensor 'goes' active** or soft motion limit is exceeded	1,4 *
Decelerate and stop the motor (under PID control) when **hard limit sensor 'goes' active** or the soft motion limit is exceeded. Use the current deceleration setting.	2,8 *
Invert the active level of the hard limit input. Typically used for normally closed hard limit sensors	128 **

* Values in red are for defining the Limit Mode for hard limits. Values in black are for defining the mode for soft motion limits. When using both hard and soft limits, parameter n should equal hard limit parameter n + soft limit parameter n.
** For inverted active level hard limits parameter n = 128 plus desired mode

```
1LM130                          ;Axis #1 Limit mode = decelerate & stop (n=2)
                                ; + invert active level(n=128)
```

LN Limits oN

MCCL command: *a*LN*n* *a* = Axis number *n* = (see table below)
compatibility: MC100, MC110
see also: LF, LM

This command is used to enable the 'hard' limit switch inputs and/or the 'soft' position limits of an axis. If a limit switch input goes active after it has been enabled by this command, and the motor has been commanded to move in the direction of that switch, the Motor Error and one of the Hard Limit Tripped Flags will be set in the motor status. At the same time the motor will be turned off or stopped (depending on the value of parameter *n* of the **L**imit **M**ode command). If a soft motion limit is enabled, and the respective axis is commanded to move beyond the motion limits set by the High motion Limit and the Low motion Limit commands, the Motor Error and one of the Soft Limit Tripped Flags will be set. At the same time the motor will be turned off or stopped (depending on the value of parameter *n* of the **L**imit **M**ode command). The flags will remain set until the motor is turned back on with the MN command. Once the motor is turned back on, it can be moved out of the limit region with any of the standard motion commands. The parameter to this command determines which of the hard and soft limits will be enabled. See the description of **Motion Limits** in the **Motion Control** chapter.

The **LN** command enables hard coded limit error checking.

Parameter n	*Hard Limits - Limit oN parameter description*
0*	Enable both hard limits (+/-) and soft limits (high & low)
1**	Enable hard limit + error checking
2**	Enable hard limit – error checking
3**	Enable hard limit + and hard limit – error checking
4**	Enable high soft limit error checking
8**	Enable low high soft limit error checking
12**	Enable high & low soft limit error checking

* If parameter n = 0 both hard and soft limit error checking will be enabled.
** Values in red are for enabling limit error checking for hard limits. if both hard and soft limits are to be used the parameter n should equal hard limit parameter n + soft limit parameter n.

```
1LN0                            ;Axis #1 - enable hard and soft limits

2LN7                            ;Axis #2 – enable both hard limits (n=3) and
                                ;high soft motion limit (n=4)
```

SA Set Acceleration

MCCL command: *a*SA*n* *a* = Axis number *n* = integer or real >= 0
compatibility: MC100, MC110
see also: DS, SV

Set the maximum acceleration rate for a given axis. The default units for the command parameter are encoder counts per second per second.

SD — **S**et **D**erivative gain

MCCL command: *a*SD*n* *a* = Axis number *n* = integer >= 0 < = 32767
compatibility: MC100, MC110
see also: FR, IL, SI, SG

This command is used to set the derivative gain of a servo's feedback loop. Increasing the derivative gain has the effect of dampening oscillations. See the description of **Tuning the Servo** in the **Motion Control** chapter.

SE — **S**top on following **E**rror

MCCL command : *a*SE*n* *a* = Axis number *n* = integer < 0 > = 32767
compatibility: MC100, MC110
see also:

Used to set the maximum following or position error for a servo (default = 1024). Once this command is issued and the motor is on, if the servo position error exceeds the specified value the motor error flag in servo status will be set, and the servo will be turned off. The error flag will remain set until the motor is turned back on with the MN command. Following error checking cannot be disabled.

SG — **S**et **P**roportional gain

MCCL command : *a*SG*n* *a* = Axis number *n* = integer >= 0 <= 32767
compatibility: MC100, MC110
see also: IL, SI, SD

This command is used to set the proportional gain of a servo's feedback loop. Increasing the proportional gain has the effect of increasing the restoring force holding a servo in position. See the description of **Tuning the Servo** in the **Motion Control** chapter.

SI — **S**et the **I**ntegral gain

MCCL command : *a*SI*n* *a* = Axis number *n* = integer >= 0 < = 32767
compatibility: MC100, MC110
see also: IL, SI, SG

The integral term accumulates the position error for servos and generates an output signal to reduce the position error to zero. The integral gain determines the magnitude of this term. The default value is zero. Note that Integration Limit (IL) command must be set to a nonzero value before integral gain will have any effect. See the description of **Tuning the Servo** in the **Motion Control** chapter.

SV — **S**et **V**elocity

MCCL command : *a*SV*n* *a* = Axis number *n* = integer or real >= 0
compatibility: MC100, MC110
see also: SA, DS

Set the maximum velocity for a given axis. The default units for the command parameter are encoder counts per second. 'On the fly' velocity changes will not take effect until after re-enabling the axis with the ***a*GO** commandfunction.

UA set the defa**U**lt **A**xis

MCCL command: UA*n* *n* = integer > 0, <= 8
compatibility: MC100, MC110
see also:
The DCX-PCI100 defaults to setting the default axis to zero. If the user executes a motion or setup command with the axis specifier missing, the default axis will be used. In most cases a motion or setup command issued to axis zero commands that operation to all axes. By defining a non-zero default axis, the user can execute 'generic' macro's (no axis number specified) to any axis.

This command is used to define a default axis. After issuing this command, any commanded move, setup, etc. command that utilizes an axis designator (*a*) will execute the command to the axis specified by parameter n. To query the controller as to the current default axis use the **G**et defa**U**lt axis (**GU**) command.

```
MD10,MR1000                    ;Macro 10 will execute a relative move
                               ;of 1000 counts to the default axis
                               ;(defined by the User Axis command).
                               ;Note that the move command does not
                               ;include the axis designator a.
UA1,MC10                       ;Define axis #1 as the default axis,
                               ;call macro ten to move 1000 counts
UA2,MC10                       ;Define axis #2 as the default axis,
                               ;call macro ten to move 1000 counts
```

UO **U**ser **O**ffset

MCCL command: aUOn *a* = Axis number *n* = integer or real
compatibility: MC100, MC110
see also:
This command is used to define a 'work area zero' position. Use parameter n to define the distance from the servo home position, to the 'work area zero' position. This offset distance must use the same usits as currently defined by the User Scaling command. This command does not take effect until after a **M**otor o**N** (***a*MN**) command. See the description of **Defining User Units** in the **Application Solutions** chapter.

UP **U**se **P**hysical axis addressing

MCCL command: *a*UP*n* *a* = Axis number *n* = integer > 0, < 8
compatibility: MC100, MC110
see also:
This command is used to reassign the axis number of a DCX motion module. The value ***a*** should equal the new axis designator. The parameter ***n*** should equal the current physical location of the motor module. Prior to reassigning axis numbers all default axis assignments must be cleared by issuing the **UP** command no values expressed for ***a*** or ***n***. See the description of **Physical Assignment of Axes Numbers** in the **Application Solutions** chapter.

UR — **U**ser **R**ate

MCCL command: *a*UR*n* *a* = Axis number *n* = integer or real >= 0
compatibility: MC100, MC110
see also:
This command is used to configure an axis for commands in user units. The default setting is 1.0. This command does not take effect until after a **M**otor o**N** (***a*MN**) command. See the description of **Defining User Units** in the **Application Solutions** chapter.

US — **U**ser **S**cale

MCCL command : *a*US*n* *a* = Axis number *n* = integer or real
compatibility: MC100, MC110
see also:
This command is used to configure an axis for commands in user units. The default setting is 1.0. This command does not take effect until after a **M**otor o**N** (***a*MN**) command. See the description of **Defining User Units** in the **Application Solutions** chapter.

UT — **U**ser **T**ime

MCCL command: *a*UT*n* *a* = Axis number *n* = integer or real >= 0
compatibility: MC100, MC110
see also:
This command is used to define the units of time for Wait commands (WA, WS, WT). The default setting is seconds. This command does not take effect until after a **M**otor o**N** (***a*MN**) command. Note – The **UT** command only effects the time base in the **task or command interface** from which it was issued. See the description of **Defining User Units** in the **Application Solutions** chapter.

UZ — set the **U**ser **Z**ero position

MCCL command: *a*UZ*n* *a* = Axis number *n* = integer or real
compatibility: MC100, MC110
see also:
This command is used to define a part program zero position. This command does not take effect until after a **M**otor o**N** (***a*MN**) command. See the description of **Defining User Units** in the **Application Solutions** chapter.

Chapter Contents

Chapter

28

MCCL Mode Commands

PM — **P**osition **M**ode

MCCL command : *a*PM *a* = Axis number
compatibility: MC100, MC110
see also: MA, MR

This command places a servo in the Position Mode of operation. In this mode, it can be commanded to execute moves to specific positions. The moves will be carried out using a trapezoidal. When in Position Mode, servos can change the move destination while the move is in progress. Upon start up, or after a Reset, motors will be placed in the Position Mode. See the description of **Point to Point Motion** in the **Motion Control** chapter.

VM — **V**elocity **M**ode

MCCL command: aVM *a* = Axis number
compatibility: MC100, MC110
see also: DI, GO

This command places a motor in the Velocity Mode of operation. In this mode, the motor can be commanded to move in either direction at a given velocity. The motor will move in that direction until commanded to stop. In Velocity Mode the user can specify the direction for the motor to move using the DIrection (DI) command. While a motor is moving, the user can issue new direction or velocity commands. The acceleration or deceleration rate at which the motor velocity will change is determined by the Set Acceleration (SA) and Deceleration Set (DS) commands. See the description of **Continuous Velocity Motion** in the **Motion Control** chapter.

Chapter Contents

Chapter

29

MCCL Motion Commands

AB — ABort motion

MCCL command: *a*AB *a* = Axis number (0 = Abort motion on all axes)
compatibility: MC100, MC110
see also: ST

This command serves as an emergency stop. For a servo, motion stops abruptly but leaves the position feedback loop (PID) and the amplifier enabled. The target position of the axis is set equal to the current position. This command can be issued to a specific axis, or can be issued to all axes simultaneously by using an axis specifier of 0.

```
example:  2AB                          ;causes the motion of axis 2 to be
                                       ;aborted
```

FI — Find Index

MCCL command: *a*FI*n* *a* = Axis number *n* = integer or real >= 0
compatibility: MC100, MC110
see also: DH, FE, IA, WI

This command is used to initialize a servo's encoder at a given position. It will remain in effect until the encoder index pulse goes active. Upon completion of the FI command, after issuing PM and MN, the position of index will be redefined *n*. This command will not start or stop any servo motions, it is up to the user to initiate motion prior to issuing the find index command. Since an index pulse may occur at numerous points of a servo's travel (once per revolution in rotary encoders), a typical servo application will require a coarse home signal to "qualify" the index pulse.

```
MD1,1LM2,1LN3,MJ10                     ;call homing macro
MD10,1VM,1DI0,1GO,LU"STATUS",1RL@0,IS25,MJ11,NO,IS17,MJ12,NO,JR-7
                                       ;test for sensors (home and +limit)
MD11,1ST,1WS.01,1DI1,1GO,1WE1,1ST,1DI0,1GO,1WE0,1FI0,1 ST,1WS.01,1PM,1MN,1MA0
                                       ;if home sensor true, initialize on
                                       ;index
MD12,1WS.1,1MN,1DI1,1GO,1WE0,MJ11      ;move negative until home true
```

See the description of **Homing Axes** in the **Motion Control** chapter.

GH — **G**o **H**ome

MCCL command: *a*GH *a* = Axis number
compatibility: MC100, MC110
see also: MA, MC, MD

Causes the specified axis or axes to move to absolute position 0. This is equivalent to a Move Absolute command, where the destination is 0.

GO — GO

MCCL command: *a*GO *a* = Axis number
compatibility: MC100, MC110
see also: CM, VM

Causes one or all axes to begin motion in velocity mode.

HO — **HO**me

MCCL command: *a*HO *a* = Axis number
compatibility: MC100, MC110
see also: MC, MD

This command will cause a user defined macro to be executed. It is up to the user to define the macro to carry out the appropriate homing sequence for that motor (see Find Edge and Find Index commands). Issuing 1HO will cause macro 1 to be executed, issuing 2HO will cause macro 2 to be executed, and so on. Issuing this command with no motor specified will cause macro 9 to be executed. See the description of **Homing Axes** in the **Motion Control** chapter.

LP — **L**earn **P**osition

MCCL command: *a*LP*n* *a* = Axis number *n* = integer >= 0, <=255
compatibility: MC100, MC110
see also: LT, MP

Used for storing the current position of one or more axes in the DCX's point memory. Positions stored in the point memory can be used by the Move to Point command to repeat a stored motion pattern. The command parameter n specifies the entry in the point memory where the position will be stored.

If the LP command is issued with an axis specifier of 0, the positions of all axes on the DCX board will be stored in the point memory. If the command is issued with a non-zero axis specifier, only the position of that axis will be stored in the point memory. No other positions in the point memory will be changed. See the description of **Learning/ Teaching Points** in the **Application Solutions** chapter.

LT — **L**earn **T**arget

MCCL command: *a*LT*n* *a* = Axis number *n* = integer >= 0, <=255
compatibility: MC100, MC110
see also: LP, MP

Similar to the LP command, but stores the axes' target position (versus actual position). Motion of an axis is not required for storing target positions. This makes it possible to download coordinates from a host computer or CAD system.

Turn off the motor drive outputs with the MF command, then send motion commands prior to the LT command. Targets stored in the point memory can be used by the Move to Point command to repeat a stored motion pattern. The command parameter n specifies the entry in the point memory where the position will be stored. If the LT command is issued with an axis specifier of 0, the targets of all axes on the DCX board will be stored in the point memory. If the command is issued with a non-zero axis specifier, only the target of that axis will be stored in the point memory. No other targets in the point memory will be changed. See the description of **Learning/ Teaching Points** in the **Application Solutions** chapter.

MA — **M**ove **A**bsolute

MCCL command : *a*MA*n* *a* = Axis number *n* = integer or real >= 0
compatibility: MC100, MC110
see also: MR, PM

This command generates a motion to absolute position *n*. A motor number must be specified and that motor must be in the ‘on’ state for any motion to occur. If the motor is in the off state, only its' internal target position will be changed. See the description of **Point to Point Motion** in the **Motion Control** chapter.

MF — **M**otor o**F**f

MCCL command : *a*MF *a* = Axis number
compatibility: MC100, MC110
see also: MN

Issuing this command will place one or all servos in the "off" state. For servos, the Analog Signal will go to the null level, the servo loop (PID) will terminate, and the Amplifier Enable output will go inactive. This command can be used to prevent unwanted motion or to allow manual positioning of the servo motor.

MN — **M**otor o**N**

MCCL command: aMN *a* = Axis number
compatibility: MC100, MC110
see also: MF

Use this command to place one or all servo motors in the on state. If an axis is off when this command is issued, the target and optimal (commanded) positions will be set to the motor's current position. This can cause a change in the axis' reported position based on new user units. At the same time, a servo module's Amplifier Enable output signal will go active. This has the effect of causing a servo to hold its current position. If an axis is already on when this command is issued, the position values will be set for the current user units, but the commanded encoder or pulse position will not be changed.

MP — **M**ove to **P**oint

MCCL command: *a*MP*n* *a* = Axis number *n* = integer >= 0, <=255
compatibility: MC100, MC110
see also: LP, LT

Used for moving one or more axes to a previously stored point. The command parameter n specifies which entry in the DCX's point memory is to be used as the destination of the move. If the MP command is issued with an axis specifier of 0, all axes will move to the positions stored in the point memory for that point. If the command is issued with a non-zero axis specifier, only that axis will move to the position in the point memory. No other axes will be commanded to move. Points can be stored in the point memory with the Learn Point (LP) and Learn Target LT) commands. See the description of **Learning/ Teaching Points** in the **Application Solutions** chapter.

MR — **M**ove **R**elative

MCCL command : *a*MR*n* *a* = Axis number *n* = integer or real
compatibility: MC100, MC110
see also: MA, PM

This command generates a motion of relative distance *n*. A motor number must be specified and that motor must be in the 'on' state for any motion to occur. If the motor is in the off state, only its' internal target position will be changed. See the description of **Point to Point Motion** in the **Motion Control** chapter.

PR — **R**ecord axis data

MCCL command : *a*PR*n* *a* = Axis number *n* = integer > 0, <=512
compatibility: MC100, MC110
see also:

This command is used to begin the recording of motion data (actual position, optimal position, and following error) for an axis. See the description of **Record and display Motion Data** in the **Application Solutions** chapter.

ST — **ST**op

MCCL command: *a*ST *a* = Axis number (0 = Stop motion on all axes)
compatibility: MC100, MC110
see also: AB, MF

This command is used to stop one or all motors. It differs from the Abort command in that motors will decelerate at their preset rate, instead of stopping abruptly. This command can be issued to a specific axis, or can be issued to all axes simultaneously by using an axis specifier of 0. See the description of **Continuous Velocity Motion** in the **Motion Control** chapter.

Chapter Contents

Chapter

30

MCCL Reporting Commands

The commands in this section are used to display the current values of internal controller data. Some of these values are 'real' numbers that must be displayed with fractional parts. In order to provide compatibility with older products that don't support real numbers, and to provide flexibility in the display format, certain reporting commands accept a parameter that sets the number of digits displayed to the right of the decimal point. These commands will show a 'p' as a parameter in their descriptions.

For ASCII command interfaces, *p* can be replaced with a number between 0 and 1 and the tenths digit will be interpreted as the number of decimal digits to display to the right of the decimal point. If no parameter is used with the command, or a parameter of 0 is used, the reply to the command will be an integer with no decimal point. Example:

```
;If axis 1 position is 123.4567
  1TP;      DCX replies 123
  1TP0;     DCX replies 123
  1TP.1;    DCX replies 123.4
  1TP.3;    DCX replies 123.456
```

For the Binary command interface, the reporting commands that have a 'p' listed as their parameter will accept an integer value of 0, 1 or 2 in place of *p*. A value of 0 will generate an integer reply, a value of 1 will generate a 64 bit floating point reply, and a value of 2 will generate a 32 bit floating point reply. See the appendix describing the DCX Binary Command Interface for more details on these reply formats.

DO — **D**isplay the recorded **O**ptimal position

MCCL command: aDOp *a* = Axis number *p* = integer >= 0, < 512
compatibility: MC100, MC110
see also: PR

This command is used to report the captured optimal position of an axis. See the description of **Record and display Motion Data** in the **Application Solutions** chapter.

DR — **D**isplay **R**ecorded position

MCCL command: aDRp *a* = Axis number *p* = integer >= 0, < 512
compatibility: MC100, MC110
see also: PR

This command is used to report the captured actual position of an axis. See the description of **Record and display Motion Data** in the **Application Solutions** chapter.

TA — **T**ell **A**nalog

MCCL command: TAx *x* = Channel number *p* = 0, 1, 2, 3, ... (# of MC500/510 modules X 4)
compatibility: MC500, MC510
see also:

Reports the digitized analog input signals to MC500 and MC510 modules. The analog input channels on any installed MC500/510 modules will be numbered sequentially starting with channel 1. For each of these channels, the TA command will display a number between 0 and 4096. These numbers are the ratio of the analog input voltage to the reference input voltage multiplied by 4096. See the description of **Analog Inputs** in the **DCX General Purpose I/O** chapter.

TB — **T**ell **B**reakpoint

MCCL command: aTBp a = Axis number *p* = 0, .1, .2, .3, .4, .5
compatibility: MC100, MC110
see also: IP, IR

Reports the position where the breakpoint for a motor is placed. Breakpoints are placed with the IP, IR, WP and WR commands. The interpretation of the command parameter p is explained at the beginning of this section.

TC — **T**ell digital **C**hannel

MCCL command: TC*x* *x* = Channel number
compatibility: MC400
see also:

Reports the on/off status of each digital I/O line. This data is reported separately for each channel. The DCX responds by displaying the channel number and a "1" if the channel is "on", or a "0" if the channel is "off".

TD — **T**ell **D**erivative gain

MCCL command: aTD a = Axis number
compatibility: MC100, MC110
see also: SD

Reports the derivative gain setting for a servo.

TE — **T**ell command interpreter **E**rror

MCCL command: TE
compatibility: Not applicable
see also:
Reports the last command interpreter error (syntax error, invalid character, etc.). For a listing of error codes please refer to the **MCCL Error Codes** chapter.

TF — **T**ell **F**ollowing error

MCCL command: *a*TF*p* a = Axis number *p* = 0, .1, .2, .3, .4, .5
compatibility: MC100, MC110
see also: SE
Reports the current following error of a servo. This error is the difference between the commanded position (calculated by the trajectory generator) and the current position.

TG — **T**ell proportional **G**ain

MCCL command: *a*TG a = Axis number
compatibility: MC100, MC110
see also: SG
Reports the proportional gain setting for a servo.

TI — **T**ell **I**ntegral gain

MCCL command: *a*TI a = Axis number 5
compatibility: MC100, MC110
see also: set Integral gain
Reports the integral gain setting for a servo.

TL — **T**ell integral **L**imit setting

MCCL command: *a*TL a = Axis number
compatibility: MC100, MC110
see also: IL
Reports the integral limit setting for a servo.

TM — **T**ell **M**acros

MCCL command: TM*n* *n* = integer >= -1, <= 1000
compatibility: N/A
see also: MD, RM
Displays the commands which make up any macros which have been defined. If n = -1, all macros will be displayed. Since macros may be defined in any sequence, the TM command is useful for confirming the existence and/or contents of macro commands. In addition to the contents of macros, this command will also show the amount of memory available for macro storage. See the description of **Macro Commands** in the **Working with MCCL Commands** chapter.

TO — **T**ell **O**ptimal

MCCL command: *a*TO*p* a = Axis number *p* = 0, .1, .2, .3, .4, .5
compatibility: MC100, MC110
see also:
Reports the desired position for servos. This value will be different than the position reported by the TP command if a following error is present.

TP — **T**ell **P**osition

MCCL command: *a*TP*p* a = Axis number *p* = 0, .1, .2, .3, .4, .5
compatibility: MC100, MC110
see also: DH, FI
Reports the absolute position of axis a. It may be used to monitor motion during both Motor oN (MN) and Motor ofF (MF) states. The interpretation of the command parameter p is explained at the beginning of this section.

TR — **T**ell **R**egister '*n*'

MCCL command: TR*n* *n* = integer >= 0, <= 255
compatibility: N/A
see also: AL, AR
Displays the contents of User Register *n*. When the command parameter is set to 0 (or not specified), this command reports the contents of User Register zero, which is the accumulator.

TS **T**ell axis **S**tatus

MCCL command : *a*TS*n* a = Axis number *n*= integer
compatibility: MC100, MC110
see also:

Reports the status of an axis. If the command parameter is 0, the response is coded into a single 32 bit value. If the parameter has a value between 1 and 31 inclusive, the state of the respective bit is displayed as a '0' for reset, and a '1' for set. Using a command parameter greater that 32 results in formatted status displays. The meaning of each bit is listed below:

Bit number	***Description***
0	Busy (motor data being updated)
1	Motor On
2	At Target
3	Trajectory Complete (Optimal = Target)
4	Direction (0 = positive, 1 = negative)
5	Reserved
6	Motor homed
7	Motor Error (Limit +/- tripped, max. following error exceeded)
8	Looking For Index (FI, WI)
9	Looking For Edge (FE, WE)
10	Index found
11	Position Capture flag
12	Breakpoint Reached (IP, IR, WP, WR)
13	Exceeded Max. Following Error
14	Servo Amplifier Driver Fault Enabled
15	Servo Amplifier Driver Fault Tripped
16	Hard Limit Positive Input Enabled
17	Hard Limit Positive Tripped
18	Hard Limit Negative Input Enabled
19	Hard Limit Negative Tripped
20	Soft Motion Limit High Enabled
21	Soft Motion Limit High Tripped
22	Soft Motion Limit Low Enabled
23	Soft Motion Limit Low Tripped
24	Encoder Index
25	Encoder Coarse home (current state)
26	Servo Amplifier Fault (current state)
27	Reserved
28	Limit Positive Input Active (current state)
29	Limit Negative Input Active (current state)
30	Reserved
31	Reserved

```
example:
   DM                              ;Place DCX in Decimal Output Mode
   1TS                             ;report the status of axis #1

  DCX returns:   01  268439566    ;status =
                                        ;bit 28 set - limit + input active
                                        ;but limits error checking is not
                                        ;enabled (bit 16 cleared)
                                        ;bit 12 set - breakpoint reached
                                        ;bit 3 set - trajectory complete
                                        ;bit 2 set - axis At target
                                        ;bit 1 set – motor on


example:  HM                            ;Place DCX in Hexadecimal Output Mode
      1TS                               ;report the status of axis #1

  DCX returns:   01  1000100E           ;status =
                                        ;bit 28 set - limit + input active
                                        ;but limits error checking is not
                                        ;enabled (bit 16 cleared)
                                        ;bit 12 set - breakpoint reached
                                        ;bit 3 set - trajectory complete
                                        ;bit 2 set - axis At target
                                        ;bit 1 set – motor on



example:       1TS32
```

DCX returns:
MOTOR STATUS:
Motor On
At Target
Trajectory Complete
Direction = Positive
Not Homed
No Motor Error
Not Looking For Index
Not Looking For Edge
Breakpoint Reached
Max. Following Error Not Exceeded
Amplifier Fault Disabled
Hard Motion Limit Positive Disabled
Hard Motion Limit Negative Disabled
Soft Motion Limit High Disabled
Soft Motion Limit Low Disabled
Index Input = 1
Coarse Home Input = 0
Amplifier Fault Input = 0
Limit Positive Input = 1
Limit Negative Input = 0
User Input 1 = 0
User Input 2 = 0

Axis Auxiliary Status

Bit number	***Description***
0	Hard Motion Limit Mode = Stop abrupt
1	Hard Motion Limit Mode = Decelerate to a stop
2	Soft Motion Limit Mode = Stop abrupt
3	Soft Motion Limit Mode = Decelerate to a stop
4	Reserved
5	Reserved
6	Reserved
7	Reserved
8	Reserved
9	Reserved
10	Reserved
11	Reserved
12	Reserved
13	Reserved
14	Reserved
15	Reserved
16	Reserved
17	Reserved
18	Reserved
19	Reserved
20	Reserved
21	Reserved
22	Positive Limit Invert = On
23	Negative Limit Invert = On
24	
25	
26	Amplifier Fault stop abrupt
27	Amplifier Fault stop smooth
28	Amplifier Fault invert input

To report the state of all Auxiliary Axis Status bits issued the Tell Status command with parameter *n* = 33:

```
example:       1TS33
```

```
example:       1TS34
```

DCX returns:
Motor status: 100100c
Auxiliary status: 20
Position count: 0
Optimal count: 0
Index count: 0
Position: 0.000000
Target: 0.000000
Optimal position: 0.000000
Break position: 0.000000
Maximum following error: 1024.000000

Motion limits: Low: 0.000000 High: 0.000000
User Scale: 1.000000
User Zero: 0.000000
User Offset: 0.000000
User Rate Conv.: 1.000000
User output constant: 1.000000
Programmed velocity: 10000.000000
Programmed acceleration: 10000.000000
Programmed deceleration: 10000.000000
Minimum velocity: 0.000000
Current velocity: 0.000000
Module ADC Input 1: 0.019530 Input2: 0.000000

TT Tell Target

MCCL command: *a*TT*p* a = Axis number *p* = 0, .1, .2, .3, .4, .5
compatibility: MC100, MC110
see also:
Reports target position. This is the absolute position to which the servo was last commanded to move. It may be specified directly with the Move Absolute (MA) command or indirectly with the Move Relative (MR) command. The interpretation of parameter *p* is explained at the beginning of this section.

TV Tell Velocity

MCCL command : aTV*p* a = Axis number *p* = 0, .1, .2, .3, .4, .5
compatibility: MC100, MC110
see also: HS, LS, MS
Reports the current velocity of a servo motor. The value is reported in units of encoder counts per servo loop update.

TZ Tell index position

MCCL command: *a*TZ*p* a = Axis number *p* = 0, .1, .2, .3, .4, .5
compatibility: MC100, MC110
see also:
Reports the position where the index pulse was observed. This position is relative to the encoder's position when the controller was reset or a Define Home command was issued to the axis.

VE tell firmware **VE**rsion

MCCL command: VE
compatibility: N/A
see also:
Reports the revision level of the firmware running on the DCX. This command also displays the amount of memory installed on the DCX motion controller motherboard.

example: VE

DCX returns:
DCX-PCI100 Motion Controller
Hardware: 4096K Private RAM, 512K Flash Memory
System Firmware Ver. PM1 Rev. 1.0a
Copyright (c) 1994-2001 Precision MicroControl Corporation
All rights reserved.

Chapter Contents

Chapter

31

MCCL I/O Commands

CF — **C**hannel o**F**f

MCCL command: CF*x* *x* = Channel number
compatibility: MC400
see also: CN

Causes channel x to go to "off" state. If the channel has been configured for "high true", the channel will be at a logic low (less that 0.4 volts DC) after this command is executed. If it has been configured for "low true", the channel will be at a logic high (greater than 2.4 volts DC).

CH — **C**hannel **H**igh

MCCL command : CH*x* *x* = Channel number or 0
compatibility: MC400
see also: CL

Causes digital I/O channel x to be configured for "high true" logic. This means that the I/O channel will be at a high logic level (greater than 2.4 volts DC) when the channel is "on". and at a low logic level (less than 0.4 volts DC) when the channel is "off". Note that issuing this command will not cause the I/O channel to change its current state. Issuing this command without specifying a channel will cause all channels present on the DCX to be configured as "high true". If parameter *x* = 0 all digital I/O channels will be configured for high true logic.

CI — **C**hannel **I**n

MCCL command: CI*x* *x* = Channel number
compatibility: MC400
see also: CT

Used to configure digital I/O channel x as an input. All digital I/O channels on the DCX default to inputs on power-on or reset. If they are subsequently changed to outputs with the Channel ouT command, they can be returned to inputs with the Channel In command. The state of a digital I/O channel can be viewed with the Tell Channel command.

CL — **C**hannel **L**ow

MCCL command : CL*x* *x* = Channel number or 0
compatibility: MC400
see also: CH

Causes digital I/O channel x to be configured for "low true" logic. This means that the I/O channel will be at a low logic level (less than 0.4 volts DC) when the channel is "on", and at high logic level (greater than 2.4 volts DC) when the channel is "off". Note that issuing this command will not cause the I/O channel to change its current state. Issuing this command without specifying a channel will cause all channels present on the DCX to be configured as "low true". If parameter *x* = 0 all digital I/O channels will be configured for low true logic.

CN — **C**hannel o**N**

MCCL command: CN*x* *x* = Channel number
compatibility: MC400
see also: CF

Causes channel x to go to "on" state. If the channel has been configured for "high true", the channel will be at a logic high (greater than 2.4 volts DC) after this command is executed. If it has been configured for "low true", the channel will be at a logic low (less that 0.4 volts DC).

CT — **C**hannel ou**T**

MCCL command: CT*x* *x* = Channel number
compatibility: MC400
see also: CI

Used to configure digital I/O channel x as an output. The DCX will turn the channel "off" before changing it to an output.

DF — **D**o if channel o**F**f

MCCL command: DF*x* x = Channel number

Used for conditional execution of commands. If digital I/O channel *x* is "off", commands that follow on the command line or in the macro will be executed. Otherwise the rest of the command line or macro will be skipped. See the description of **Digital I/O** in the **DCX General Purpose I/O** chapter.

```
DF2,1MR1000                          ;If channel 2 is off move 1000
```

DN — **D**o if channel ‘x’ is o**N**

MCCL command: DN*x* x = Channel number

Used for conditional execution of commands. If digital I/O channel *x* is "on", commands that follow on the command line or in the macro will be executed. Otherwise the rest of the command line or macro will be skipped. See the description of **Digital I/O** in the **DCX General Purpose I/O** chapter.

```
DN2,1MR1000                          ;If channel 2 is off move 1000
```

GA — **G**et **A**nalog

MCCL command: GA*x* x = Channel number
compatibility: MC500, MC520

Performs analog to digital conversion on the specified input channel and places the result into the Accumulator (User Register 0). Analog channels are numbered starting with 1.

IF — **I**f channel o**F**f do next command, else skip 2 commands

MCCL command: IF*x* x = Channel number

Used for conditional execution of commands. If digital I/O channel *x* is "off", command execution will continue with the command following the IF command. Otherwise the two commands following the IF command will be skipped, and command execution will continue from the third command. See the description of **Digital I/O** in the **DCX General Purpose I/O** chapter.

```
IF5,MJ10,NO,MJ11                    ;If digital input #5 is off jump to
                                    ;macro 10, otherwise jump to macro 11
```

IN — **I**f channel ' o**N** do next command, else skip 2 commands

MCCL command: IN*x* x = Channel number

Used for conditional execution of commands. If digital I/O channel *x* is "on", command execution will continue with the command following the IN command. Otherwise the two commands following the IN command will be skipped, and command execution will continue from the third command. See the description of **Digital I/O** in the **DCX General Purpose I/O** chapter.

```
IN5,MJ10,NO,MJ11                    ;If digital input #5 is on jump to
                                    ;macro 10, otherwise jump to macro 11
```

OA — **O**utput **A**nalog

MCCL command: OA*n* *n* = integer or real
compatibility MC500, MC520

Sets the specified analog output channel to the value stored in the Accumulator (User Register 0). The analog output channels on any installed MC500 modules are numbered consecutively starting with channel 1. The contents of the Accumulator should be in the range 0 to 4095.

TA — **T**ell **A**nalog

MCCL command: TAx *x* = Channel number *p* = 0, 1, 2, 3, ... (# of MC500/510 modules X 4)
compatibility: MC500, MC510
see also:

Reports the digitized analog input signals to MC500 and MC510 modules. The analog input channels on any installed MC500/510 modules will be numbered sequentially starting with channel 1. For each of these channels, the TA command will display a number between 0 and 4096. These numbers are

the ratio of the analog input voltage to the reference input voltage multiplied by 4096. See the description of **Analog Inputs** in the **DCX General Purpose I/O** chapter.

TC **T**ell **C**hannel

MCCL command: TC*x* *x* = Channel number or 0
compatibility: MC400
see also:
Reports the on/off status of each digital I/O line. This data is reported separately for each channel. The DCX responds by displaying the channel number and a "1" if the channel is "on", or a "0" if the channel is "off". If parameter *x* = 0 the state of all digital I/O channels will reported.

Chapter Contents

Chapter

32

MCCL Macro and Multi-tasking Commands

BK — BreaK

MCCL command: BK
see also: GT, TR

Execution of this command will cause the rest of the command line or macro to be skipped. This command is used in conjunction with the If oN and If ofF commands to implement conditional execution.

ET — Escape Task

MCCL command: ET*n* *n* = integer > = 0
see also: GT, TR

This command is used to terminate a 'background task' that was created with the Generate Task command. The parameter to this command must be the task identifier that was placed in the accumulator (user register 0) of the task that issued the Generate Task command. A background task can use this command to terminate itself, but it must first acquire its identifier from the 'parent' task through a global register. Note that the task that interprets and executes commands received from the command interfaces cannot be terminated. See the description of **Multi-Tasking** in the **Command Set Introduction** chapter.

GT — Generate Task

MCCL command: GT*n* *n* = integer > = 0, < = 1000
see also: ET, MC, MD, TR

This command will cause macro *n* to be executed as a background task. Alternatively, this command can precede a sequence of commands. In this case, the commands following the Generate Task command will be executed as a background task. After this command is issued, an identifier for the background task will be placed in the accumulator (register 0) of the task that issued the command. This identifier can be used as the parameter to the Escape Task command to terminate the background task. See the description of **Multi-Tasking** in the **Command Set Introduction** chapter.

MC **M**acro **C**all

MCCL command: MC*n* *n* = integer > = 0, < = 1000
see also: ET, MD

This command may be used to execute a previously defined macro command. If there is no macro defined by the number n, an error message will be displayed. Macro Call Commands can also be used in compound commands with other commands in the instruction set. In addition, a macro command can call another macro command, which in turn can call another macro command, and so on. See the description of **Building MCCL Macro Sequences** in the **Command Set Introduction** chapter.

MD Macro Define

MCCL command: MD*n* *n* = integer > = 0, < = 1000
see also: ET, GT, MD, RM

Used to define a new macro. This is done by placing the Macro Define command as the first command in a sequence of commands. All commands following the Macro Define command will be included in the macro. See the description of **Building MCCL Macro Sequences** in the **DCX Operation** chapter.

Macros will erased if power to the board is turned off. A macro can be redefined but the memory space occupied by the previous version of the macro will not be reused until a Reset Macro command is issued. Thus, if macro *n* already exists when a Macro Define command for that macro is issued, the previously defined macro will be replaced by the new macro definition.

MJ **M**acro **J**ump

MCCL command: MJ*n* *n* = integer > = 0, < = 1000
see also: ET, GT, MD

Jumps to a previously defined macro. This command differs from the Macro Call command in that execution will not return to the command following the MJ command. See the description of **Building MCCL Macro Sequences** in the **DCX Operation** chapter.

NO **N**o **O**peration

MCCL command: NO

This command does nothing. It can be used to cause short delays in command line executions or as a filler in sequence commands.

Reset Macros

MCCL command: RM

This command will initialize the memory space used for storage of macro commands. It has the effect of erasing currently defined macros from memory. It is also the only way in which macro commands can be removed from memory after they are defined. It is always a good idea to use the Reset Macro command (RM) before setting up a new set of macro commands. See the description of **Building MCCL Macro Sequences** in the **Working with MCCL Commands** chapter.

TM **T**ell **M**acros

MCCL command: TM*n* *n* = integer >= -1, <= 1000
see also: MD, RM

Displays the commands which make up any macros which have been defined. If n = -1, all macros will be displayed. Since macros may be defined in any sequence, the TM command is useful for confirming the existence and/or contents of macro commands. In addition to the contents of macros, this command will also show the amount of memory available for macro storage, both in RAM and FLASH memory. See the description of **Building MCCL Macro Sequences** in the **DCX Operation** chapter.

Chapter Contents

Chapter

33

MCCL Register Commands

AA **A**ccumulator **A**dd

MCCL command: AAn *n* = integer or real

Performs ACC = ACC + *n*, the addition of the command parameter ***n*** to the Accumulator (User Register 0). If the command parameter is in integer format, the result is stored in the Accumulator as a 32 bit integer. If the command parameter is in real format, the result is stored in the Accumulator (User Register 0 and 1) as a 64 bit real value.

AC **A**ccumulator **C**omplement, bit wise

MCCL command: AC

Performs ACC = !ACC, the bit wise logical complement of the Accumulator (User Register 0). The result is stored in the Accumulator as a 32 bit integer.

AD **A**ccumulator **D**ivide

MCCL command: ADn *n* = integer or real

Performs ACC = ACC/*n*, the division of the Accumulator (User Register 0) by the command parameter. If the command parameter is in integer format, the result is stored in the Accumulator as a 32 bit integer. If the command parameter is in real format, the result is stored in the Accumulator (User Register 0 and 1) as a 64 bit real value. No operation is done if the command parameter is zero.

AE **A**ccumulator logical **E**xclusive or with '*n*' , bit wise

MCCL command: AE*n* *n* = integer or real

Performs ACC = ACC ^ *n*, the bit wise logical exclusive or'ing of the Accumulator (User Register 0) with the command parameter. The result is stored in the Accumulator as a 32 bit integer.

AL **A**ccumulator load

MCCL command: AL*n* *n* = integer or real

Loads the Accumulator (User Register 0) with *n* . If the command parameter is an integer (no decimal point or exponent label) the Accumulator will be marked as containing a 32 bit integer, otherwise it will be marked as containing a 64 bit real value.

```
AL1234567890                          ;Load 1234567890 into the accumulator
AL1234.56789                          ;Load 1234.56789 into the accumulator
AL0.123456789e4                       ;Load 1234.56789 into the accumulator
```

AM **A**ccumulator **M**ultiply

MCCL command: AM*n* *n* = integer or real

Performs ACC = ACC * *n*, the multiplication of the Accumulator (User Register 0) by the command parameter. If the command parameter is in integer format, the result is stored in the Accumulator as a 32 bit integer. If the command parameter is in real format, the result is stored in the Accumulator (User Register 0 and 1) as a 64 bit real value.

AN **A**ccumulator logical ‘a**N**d’ the ‘*n’* , bit wise

MCCL command: AN*n* *n* = integer or real

Performs ACC = ACC & *n*, the bit wise logical AND of the Accumulator (User Register 0) with the command parameter. The result is stored in the Accumulator as a 32 bit integer.

AO **A**ccumulator logical ‘**O**r’ with ‘*n’* , bit wise

MCCL command : AO*n* *n* = integer or real

Performs ACC = ACC | *n*, the bit wise logical OR of the Accumulator (User Register 0) with the command parameter. The result is stored in the Accumulator as a 32 bit integer.

AR copy **A**ccumulator to **R**egister

MCCL command: ARn *n* = integer or real

Copies the contents of the Accumulator (User Register 0) to the User Register specified by *n*. The contents of the Accumulator are unaffected by this command.

AS **A**ccumulator **S**ubtract

MCCL command: AS*n* *n* = integer or real

Performs ACC = ACC - *n*, the subtraction of the command parameter from the Accumulator (User Register 0). If the command parameter is in integer format, the result is stored in the Accumulator as a 32 bit integer. If the command parameter is in real format, the result is stored in the Accumulator (User Register 0 and 1) as a 64 bit real value.

AV **A**ccumulator e**V**aluate

MCCL command: AVn *n* = integer >= 0, <=25

Performs a unary operation on the contents of the Accumulator (User Register 0), placing the result in the Accumulator, overwriting the original contents. Parameter *n* specifies the desired operation. The table below list the available operations and the respective command parameter to use. The result that is stored in the Accumulator (1<= *n* <= 25) will be a 64 bit real in all cases except the Convert to ASCII operation which returns an integer.

Parameter n =	***Operation***	***Return type***
1	Convert to ASCII (Address placed in ACC)	Integer
2	Change Sign	Double
3	Absolute Value	Double
4	Ceiling	Double
5	Floor	Double
6	Fraction	Double
7	Round	Double
8	Square	Double
9	Square Root	Double
10	Sine	Double
11	Cosine	Double
12	Tangent	Double
13	Arc Sine	Double
14	Arc Cosine	Double
15	Arc Tangent	Double
16	Hyperbolic Sine	Double
17	Hyperbolic Cosine	Double
18	Hyperbolic Tangent	Double
19	Exponent	Double
20	Log	Double
21	Log10	Double
22	Load Pi	Double
23	Load 2 * Pi	Double
24	Load Pi/2	Double
25	Convert double register contents to an integer	Integer

GA **G**et **A**nalog

MCCL command: GA*x* x = Channel number

Performs analog to digital conversion on the specified input channel and places the result into the Accumulator (User Register 0). Analog channels are numbered starting with 1.

GD — **G**et the module i**D**

MCCL command: GD*x* x= integer > 0, <= 8

Loads the accumulator with the type of motor module associated with an axis number

Module Type	ID code
MC100	5
MC110	4

GU — **G**et defa**U**lt axis

MCCL command: GU

The DCX-PCI100 defaults to setting the default axis to zero. If the user executes a motion or setup command with the axis specifier missing, the default axis will be used. In most cases a motion or setup command issued to axis zero commands that operation to all axes. By defining a non-zero default axis, the user can execute 'generic' macro's (no axis number specified) to any axis.
This Get default axis is used to report the current default axis by placing the current setting into the accumulator. The default axis is defined by using the setup command set the defa**U**lt **A**xis (**UA*n***).

LU — **L**ook **U**p variable

MCCL command: LUs s= *string parameter ("variable name")*

Loads the accumulator with the memory location for a motor table data entry. For additional information including a complete listing of variable names please refer to the description of **Reading Data from DCX Memory** in **Chapter 20** of this manual.

OA — **O**utput **A**nalog

MCCL command: OAx x = integer or real
compatibility MC500, MC520

Sets the analog output of channel *x* to the value stored in the Accumulator (User Register 0). The analog output channels on any installed MC500 modules are numbered consecutively starting with channel 1. The contents of the Accumulator should be in the range 0 to 4095.

RA — copy **R**egister to **A**ccumulator

MCCL command: RAn *n* = integer or real

Copies the contents of the User Register *n* into the Accumulator (User Register 0). The original contents of the accumulator is overwritten, while the contents of the source User Register are unaffected.

RB — **R**ead the **B**yte at absolute memory location '*n*' into the accumulator

MCCL command: *a*RB*n* *a* = Axis number *n* = integer

This command will copy the contents of the byte located at absolute memory address *n* into the Accumulator (User Register 0). Alternatively, if an axis number is specified with the command, the contents of a byte located within that axes' motor table will be copied into the accumulator. In this case the command parameter specifies the offset of the byte from the beginning of that axes motor table. The **Reading DCX Memory** section of this chapter lists the offsets of all data in the motor tables. The upper bits of the Accumulator are cleared when the byte data is copied into it.

RD — **R**ead the **Double** (64 bit real) value at absolute memory location '*n*' into the accumulator

MCCL command: *a*RD*n* *a* = Axis number *n* = real

This command will copy the contents of the Double (64 bit real) located at absolute memory address *n* into the Accumulator (User Register 0). Alternatively, if an axis number is specified with the command, the contents of a Double located within that axes' motor table will be copied into the accumulator. In this case the command parameter specifies the offset of the Double from the beginning of that axes motor table. The **Reading DCX Memory** section of this chapter lists the offsets of all data in the motor tables.

RL — **R**ead the **L**ong (32 bit integer) value at absolute memory location '*n*' into the accumulator

MCCL command: *a*RL*n* *a* = Axis number *n* = integer

This command will copy the contents of the Long (32 bit integer) located at absolute memory address n into the Accumulator (User Register 0). Alternatively, if an axis number is specified with the command, the contents of a Long located within that axes' motor table will be copied into the accumulator. In this case the command parameter specifies the offset of the Long from the beginning of that axes motor table. The **Reading DCX Memory** section of this chapter lists the offsets of all data in the motor tables.

RV — **R**ead the float (32 bit real) value at absolute memory location '*n*' into the accumulator

MCCL command: *a*RV*n* *a* = Axis number *n* = real

This command will copy the contents of the Float (32 bit real) located at absolute memory address *n* into the Accumulator (User Register 0). Alternatively, if an axis number is specified with the command, the contents of a Float located within that axes' motor table will be copied into the accumulator. In this case the command parameter specifies the offset of the Float from the beginning of that axes motor table. The **Reading DCX Memory** section of this chapter lists the offsets of all data in the motor tables.

RW — **R**ead the **W**ord (16 bit integer) value at absolute memory location '*n*' into the accumulator

MCCL command: *a*RW*n* *a* = Axis number *n* = integer

This command will copy the contents of the Word (16 bit integer) located at absolute memory address *n* into the Accumulator (User Register 0). Alternatively, if an axis number is specified with the command, the contents of a Word located within that axes' motor table will be copied into the accumulator. In this case the command parameter specifies the offset of the Word from the beginning of that axes motor table. The **Reading DCX Memory** section of this chapter lists the offsets of all data in the motor tables.

SL — **S**hift **L**eft accumulator by '*n*' bits

MCCL command: SL*n* *n* = integer > 0, < = 31

Performs ACC = ACC << *n*, the logical shift of the Accumulator (User Register 0) to the left. The command parameter specifies the number of bits to shift the accumulator. Zero bits will be shifted in on the right. The result is stored in the Accumulator as a 32 bit integer.

SR — **S**hift **R**ight accumulator by '*n*' bits

MCCL command: SR*n* *n* = integer > 0, < = 31

Performs ACC = ACC >> *n* , the logical shift of the Accumulator (User Register 0) to the right. The command parameter specifies the number of bits to shift the accumulator. Zero bits will be shifted in on the left. The result is stored in the Accumulator as a 32 bit integer.

TR — **T**ell **R**egister '*n*'

MCCL command: TR*n* *n* = integer >= 0, <= 256
compatibility: N/A
see also: AL, AR

Displays the contents of User Register *n*. When the command parameter is set to 0 (or not specified), this command reports the contents of User Register zero, which is the accumulator.

Chapter Contents

Chapter

34

MCCL Sequence (If/Then) Commands

DF — **D**o if channel o**F**f

MCCL command: DF*x* x = Channel number

Used for conditional execution of commands. If digital I/O channel *x* is "off", commands that follow on the command line or in the macro will be executed. Otherwise the rest of the command line or macro will be skipped. See the description of **Digital I/O** in the **DCX General Purpose I/O** chapter.

```
DF2,1MR1000                    ;If channel 2 is off move 1000
```

DN — **D**o if channel 'x' is o**N**

MCCL command: DN*x* x = Channel number

Used for conditional execution of commands. If digital I/O channel *x* is "on", commands that follow on the command line or in the macro will be executed. Otherwise the rest of the command line or macro will be skipped. See the description of **Digital I/O** in the **DCX General Purpose I/O** chapter.

```
DN2,1MR1000                    ;If channel 2 is off move 1000
```

IB — **I**f the accumulator is **B**elow 'n', execute the next command, else skip 2 commands

MCCL command: IBn *n* = integer or real

Used for conditional execution of commands. If the contents of the accumulator (User Register 0) is less than *n*, command execution will continue with the command following the IB command. Otherwise the two commands following the IB command will be skipped, and command execution will continue from the third command. See the description of **Digital I/O** in the **DCX General Purpose I/O** chapter.

```
IB0,MJ10,NO,MJ11               ;If the accumulator contents is less
                               ;than 10 jump to macro 10, otherwise
                               ;jump to macro 11
```

IC — If bit ‘n’ of the accumulator is **C**lear (equal to 0), execute the next command, else skip 2 commands

MCCL command: IC*n* *n* = integer >= 0, <= 31

Used for conditional execution of commands. If the contents of the accumulator (User Register 0) has bit *n* reset, command execution will continue with the command following the IC command. Otherwise the two commands following the IC command will be skipped, and command execution will continue from the third command.

```
IC3,MJ10,NO,MJ11                    ;If accumulator bit 3 is cleared jump
                                    ;to macro 10, otherwise jump to macro
                                    ;11
```

IE — If the accumulator **E**quals “n”, execute the next command, else skip 2 commands

MCCL command: IE*n* *n* = integer or real

Used for conditional execution of commands. If the contents of the accumulator (User Register 0) equals *n*, command execution will continue with the command following the IE command. Otherwise the two commands following the IE command will be skipped, and command execution will continue from the third command.

```
IE0,MJ10,NO,MJ11                    ;If accumulator contents equals 0 jump
                                    ;to macro 10, otherwise jump to macro
                                    ;11
```

IF — If channel o**F**f do next command, else skip 2 commands

MCCL command: IF*x* x = Channel number

Used for conditional execution of commands. If digital I/O channel *x* is "off", command execution will continue with the command following the IF command. Otherwise the two commands following the IF command will be skipped, and command execution will continue from the third command. See the description of **Digital I/O** in the **DCX General Purpose I/O** chapter.

```
IF5,MJ10,NO,MJ11                    ;If digital input #5 is off jump to
                                    ;macro 10, otherwise jump to macro 11
```

IG — If the accumulator is **G**reater than ‘n’ execute the next command, else skip 2 commands

MCCL command: IG*n* *n* = integer or real

Used for conditional execution of commands. If the contents of the accumulator (User Register 0) is greater than *n*, command execution will continue with the command following the IG command. Otherwise the two commands following the IG command will be skipped, and command execution will continue from the third command. See the description of **Digital I/O** in the **DCX General Purpose I/O** chapter.

```
IG0,MJ10,NO,MJ11                    ;If the accumulator contents is
                                    ;greater than 0 jump to macro 10,
                                    ;otherwise jump to macro 11
```

IN — **I**f channel ' o**N** do next command, else skip 2 commands

MCCL command: IN*x* x = Channel number

Used for conditional execution of commands. If digital I/O channel *x* is "on", command execution will continue with the command following the IN command. Otherwise the two commands following the IN command will be skipped, and command execution will continue from the third command. See the description of **Digital I/O** in the **DCX General Purpose I/O** chapter.

```
IN5,MJ10,NO,MJ11                    ;If digital input #5 is on jump to
                                    ;macro 10, otherwise jump to macro 11
```

IP — **I**nterrupt (set breakpoint reached flag) on absolute **P**osition

MCCL command: IP*n* *n* = integer or real

compatibility: MC100, MC110

This command is used to indicate when an axis has reached a specific position. The position is specified by parameter *n* as a relative distance from the axis home position. When the specified position has been reached, the DCX will set the "breakpoint reached" flag in the motor status for that axis. The IP command can be issued to an axis before or after it has been commanded to move.

IR — **I**nterrupt (set breakpoint reached flag) upon reaching **R**elative position

MCCL command: IR*n* *n* = integer or real

compatibility: MC100, MC110

This command is used to indicate when an axis has reached a specific position. The position is specified by parameter *n* as a relative distance from the target position established by the last motion command. When the specified position has been reached, the DCX will set the "breakpoint reached" flag in the status for that axis. The IR command can be issued to an axis before or after it has been commanded to move.

IS — **I**f bit 'n' of the accumulator is **S**et execute the next command, else skip 2 commands

MCCL command: IS*n* *n* = integer >= 0, <= 31

Used for conditional execution of commands. If the contents of the accumulator (User Register 0) has bit *n* set, command execution will continue with the command following the IS command. Otherwise the two commands following the IS command will be skipped, and command execution will continue from the third command.

```
IS3,MJ10,NO,MJ11                    ;If accumulator bit 3 is set jump to
                                    ;macro 10, otherwise jump to macro 11
```

IU — **I**f the accumulator is **U**nequal to "n" execute the next command, else skip 2 commands

MCCL command: IU*n* *n* = integer or real

Used for conditional execution of commands. If the contents of the accumulator (User Register 0) does not equal *n*, command execution will continue with the command following the IU command. Otherwise the two commands following the IU command will be skipped, and command execution will continue from the third command.

```
IU0,MJ10,NO,MJ11                    ;If accumulator contents is unequal to
                                    ;0 jump to macro 10, otherwise jump to
                                    ;macro 11
```

JP — **J**um**P** to command absolute

MCCL command: JP*n* *n* = integer

Jumps to the specified command in the current command string or macro. Commands are numbered consecutively starting with 0.

```
IE0,JP5,NO,1MR1000,1WS,1MR2000,1WS  ;If accumulator equals 0 jump to
                                    ;1MR2000
```

JR — **J**ump to command **R**elative

MCCL command: JRn *n* = integer

Jumps forward or backward by *n* commands in the current command string or macro. Specifying a positive value will cause a forward jump in the command string or macro. Specifying a negative value will cause a backward jump. A jump of relative 0 will cause the command to jump to itself.

```
1MR1000,1WS.005,IE0,JR-3            ;If accumulator equals 0 jump to
                                    ;1MR1000
```

RP — **R**e**P**eat

MCCL command: RPn *n* = integer > = 0, < = 2,147,483,647

This command causes all the commands preceding the RP command to be executed *n* + 1 times. If *n* is not specified or is 0 then the commands are repeated indefinitely. Note - There can be only one RP command in a command string or macro.

```
TP,RP999                            ;Display the position of axis #1, 1000
                                    ;times
```

WA — **WA**it

MCCL command: WAn *n* = integer or real >= 0

Insert a wait period of *n* seconds before going on to the next command. If this command was issued from an ASCII interface, it can be aborted by sending an Escape character.

```
1TP,WA0.1,RP9                    ;Display the position of axis #1, 10
                                 ;times with a delay of one tenth of a
                                 ;second between displays
```

WE — **W**ait for **E**dge

MCCL command: *a*WEx x = 0 or 1
compatibility: MC100, MC110
see also: FE

Wait until the coarse home input of a servo is at the specified logic level, and then continue operation. If x is not specified or is 0, wait for coarse home to go active. If *x* is 1 wait, for coarse home to go inactive. If this command was issued from an ASCII interface, it can be aborted by sending an Escape character.

WF — **W**ait for digital channel o**F**f

MCCL command: WF*x* x = Channel number
compatibility: MC400
see also: WN

Wait until digital I/O channel x is "off" before continuing to the next command on the command line or in the macro. If this command was issued from an ASCII interface, it can be aborted by sending an Escape character.

WI — **W**ait for encoder **I**ndex mark

MCCL command: *a*WIn *a* = Axis number *n* = integer or real >= 0
compatibility: MC100, MC110
see also: FI

Wait until the index pulse has been observed on servo axis *a*. This command should be used after a Index Arm command has been issued to the axis, even if it is known that the index pulse has occurred (this command performs internal operations). To complete the indexing function, a Motor On (aMN) command should also be issued to axis *a* to re-initialize the position registers to *n*. If this command was issued from an ASCII interface, it can be aborted by sending an Escape character.

WN — **W**ait for digital channel o**N**

MCCL command: WN*x* x = Channel number
compatibility: MC400
see also: WF

Wait until digital I/O channel x is "on" before continuing to the next command on the command line or in the macro. If this command was issued from an ASCII interface, it can be aborted by sending an Escape character.

WP — **W**ait for absolute **P**osition

MCCL command: *a*WP*n* *n* = integer or real
compatibility: MC100, MC110
see also:
This command is used to delay command execution until axis *a* has reached a specific position. The position is specified by the command parameter as a relative distance from the home position of the axis. When the specified position has been reached, the DCX will set the "breakpoint reached" flag in the status for that axis, and then continue execution of commands following WP. The WP command will typically be issued to an axis after it has been commanded to move. If this command was issued from an ASCII interface, it can be aborted by sending an Escape character.

WR — **W**ait for **R**elative position

MCCL command: *a*WR*n* *n* = integer or real
compatibility: MC100, MC110
see also:
This command is used to delay command execution until axis *a* has reached a specific position. The position is specified by the command parameter as a relative distance from the target position established by the last motion command. When the specified position has been reached, the DCX will set the "breakpoint reached" flag in the status for that axis, and then continue execution of commands following WR. The WR command will typically be issued to an axis after it has been commanded to move. If this command was issued from an ASCII interface, it can be aborted by sending an Escape character.

WS — **W**ait for **S**top

MCCL command: *a*WS*n* *n* = integer or real
compatibility: MC100, MC110
see also: Wait (a period of time), Wait for target reached
Will delay execution of the next command in the sequence until the trajectory generator for axis *a* (or all axes if axis specifier a = 0) has completed the current motion. The command parameter *n* specifies an additional time period (in seconds) that the controller will wait before continuing execution of the commands following WS.

```
3MR1000,WS0.1,MR-1000                ;Perform a forward then backward
                                     ;motion sequence
```

comment: If the WS command was not used in the above example, there would be no motion of the axis. The reason being that the target position would simply be changed twice. The computer would add 1000 counts to the target position then subtract the same amount. This would take place far quicker than the axis could begin moving.

WT — **W**ait for **T**arget

MCCL command: *a*WT*n* *n* = integer or real
compatibility: MC100, MC110
see also: WA, WS

This command will delay command execution until axis *a* (or all axes if axis specifier a = 0) has reached its target position. Parameter *n* specifies an additional time period (in seconds) that the controller will wait before continuing execution of the commands following WT. The conditions for a servo to have reached its' target, is that it remains within the position DeadBand for the time period specified by the Delay at Target parameter ‘n’.

```
3MR1000,WT0.1,MR-1000                ;Perform a forward then backward
                                     ;motion sequence
```

comment: If the WT command was not used in the above example, there would be no motion of the axis. The reason being that the target position would simply be changed twice. The computer would add 1000 counts to the target position then subtract the same amount. This would take place far quicker than the axis could begin moving.

Chapter Contents

Chapter

35

Miscellaneous Commands

DM **D**ecimal **M**ode

MCCL command: DM
see also: HM
Input and output numbers in decimal format.
comment: The Decimal Mode command must be "executed" by the DCX before commands can be issued with decimal formatted parameters. The Decimal Mode (DM) and Hexadecimal Mode (HM) commands cannot be in the same command string.

DW **D**isable **W**atchdog

MCCL command: DM
see also:
Disable the processor watchdog circuit.
comment: This command is reserved for factory use only.

FD **O**utput text with **D**oubles

MCCL command: FDs s = *string parameter*
see also: FT, OD, OT
This command places a formatted message string and double precision values into DCX memory. Upon completion of this command the memory address where the formatted message is stored is available in the accumulator (register 0). For additional information please refer to the description of **Outputting Formatted Message Strings** in **Chapter 6**.

FT **O**utput **T**ext with integers

MCCL command: FDs s = *string parameter*
see also: FD, OD, OT
This command places a formatted message string and integer values into DCX memory. Upon completion of this command the memory address where the formatted message is stored is available

in the accumulator (register 0). For additional information please refer to the description of **Outputting Formatted Message Strings** in **Chapter 6**.

HE display the supported MCCL commands

MCCL command: HE
explanation: Reports the valid DCX command mnemonics for the installed software version.

HM **H**exadecimal **M**ode

MCCL command: HM
see also: DM
Input and output numbers in hexadecimal format.
comment: The Hexadecimal Mode command must be executed by the DCX before commands can be issued with hexadecimal formatted parameters. The Hexadecimal Mode (HM) and Decimal Mode (DM) and commands cannot be in the same command string. If a command parameter is to be entered in hexadecimal format, and the number starts with either A, B, C, D, E, or F, it must be preceded by a '0' (zero).

NO **N**o **O**peration

MCCL command: NO
This command does nothing. It can be used to cause short delays in command line executions or as a filler in sequence commands.

OD **O**utput text with **D**oubles

MCCL command: ODs s = *string parameter*
see also: FD, FT, OT
This command allows the user to send formatted message strings and double precision values to the ASCII interface (WinControl). For additional information please refer to the description of **Outputting Formatted Message Strings** in **Chapter 6**.

OT **O**utput **T**ext with integers

MCCL command: OTs s = *string parameter*
see also: FD, FT, OD
This command allows the user to send formatted message strings and integer values to the ASCII interface (WinControl). For additional information please refer to the description of **Outputting Formatted Message Strings** in **Chapter 6**.

RT ReseT

MCCL command: *a*RT *a* = Axis number (0 resets the entire controller)
compatibility: MC100, MC110 , MC400, MC5X0
see also: Default Settings in the Appendix

Performs a reset of the entire controller or a specific axis. If an axis number is specified when the command is issued, just that axis will be reset. If no axis is specified, the entire controller and all installed axes will be reset. When an axis is reset, the default conditions such as acceleration and velocity will be restored, and the axes will be placed in the "off" state.

Chapter Contents

Chapter

36

MCCL Error Codes

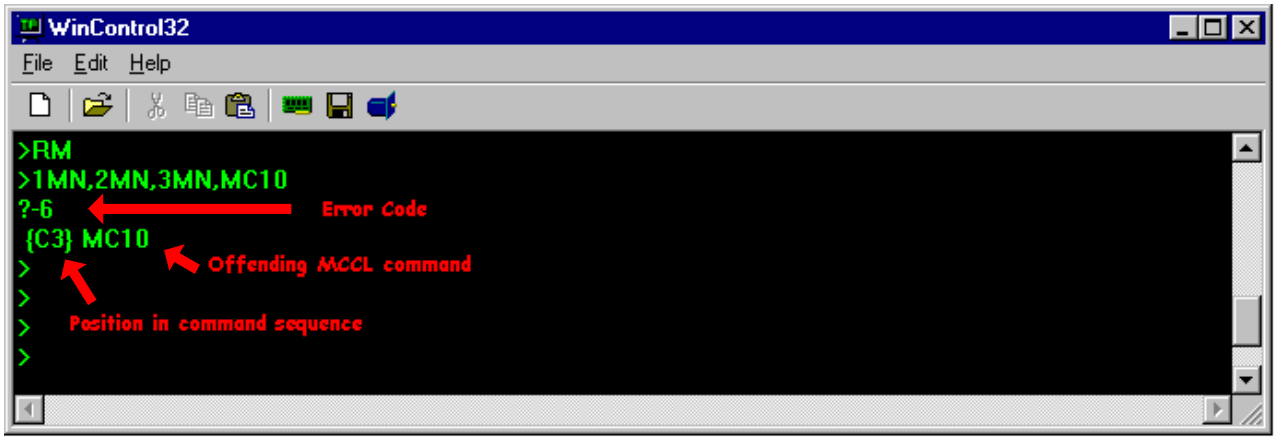
Both the MCAPI and the Motion Control Command Language (MCCL) provide error code and interface status information to the user.

MCCL Error Codes

When executing MCCL (Motion Control Command Language) command sequences the command interpreter will report the following error code when appropriate:

Description	Error code
No error	0
Unrecognized command	1
Bad command format	2
I/O error	3
Command string to long	4
Command Parameter Error	-1
Command Code Invalid	-2
Negative Repeat Count	-3
Macro Define Command Not First	-4
Macro Number Out of Range	-5
Macro Doesn't Exist	-6
Command Canceled by User	-7
	-8
	-9
	-10
No axis specified	-14
Axis not assigned	-15
Axis already assigned	-16
Axis duplicate assigned	-17

Many error code reports will not only include the error code but also the offending command. In the following example the Reset Macro command was issued. This command clears all macro's from memory. The next command sequence turns on 3 motors and then calls macro 10. The command MC10 is a valid command but with no macros in memory error code –6 is displayed.



Chapter Contents

Chapter
37

Printing a PDF Document

Introduction to PDF
PDF stands for Portable Document Format. It is the defacto standard for transporting electronic documents. PDF files are based on the PostScript language imaging model. This enables sharp, color-precise printing on almost all printers.

Printing a complete PDF document
It is **not recommended** that large PDF documents be printed on personal computer printers. The 'wear and tear' incurred by these units, coupled with the difficulties of two sided printing, typically resulting in degraded performance of the printer and a whole lot of wasted paper. PMC recommends that PDF document be printer by a full service print shop that uses digital (computer controlled) copy systems with paper collating/sorting capability.

Printing selected pages of a PDF document
While viewing a PDF document with Adobe Reader (or Adobe Acrobat), any page or range of pages can be printed by a personal computer printer by:

Selecting the printer icon on the tool bar
Selecting **Print** from the Adobe **File** menu

Paper
The selection of the paper type to be used for printing a PDF document should be based on the target market for the document. For a user's manual with extensive graphics that is printed on both sides of a page the minimum recommended paper type is 24 pound. A heavier paper stock (26 – 30 pound) will reduce the 'bleed through' inherent with printed graphics. Typically the front and back cover pages are printed on heavy paper stock (50 to 60 pound).

Binding
Unlike the binding of a book or catalog, a user's manual distributed in as a PDF file will typically use 'comb' or 'coil' binding. This service is provided by most full service print shops. Coil binding is

suitable for documents with no more than 100 pieces of paper (24 pound). Comb binding is acceptable for documents with as many as 300 pieces of paper (24 pound). Most print shops stock a wide variety of ‘combs’. The print shop can recommend the appropriate ‘comb’ based on the number of pages.

Pricing
The final cost for printing and binding a PDF document is based on:

- Quantity per print run
- Number of pages
- Paper type

The price range for printing and binding a PDF document similar to this user manual will be $15 to $30 (printed in Black & White) in quantities of 1 to 10 pieces.

Obtaining a Word 2000 version of this user manual
This user document was written using Microsoft’s Word 2000. Qualified OEM’s, Distributors, and Value Added Reps (VAR’s) can obtain a copy of this document for

- Editing
- Customization
- Language translation.

Please contact Precision MicroControl to obtain a Word 2000 version of this document.

Chapter

38

Glossary

Accuracy - A measure of the difference between the expected position and actual position of a motion system.

Actuator - Device which creates mechanical motion by converting energy to mechanical energy.

Axis Phasing - An axis is properly phased when a commanded move in the positive direction causes the encoder decode circuitry of the controller to increment the reported position of the axis.

Back EMF - The voltage generated when a permanent magnet motor is rotated. This voltage is proportional to motor speed and is present regardless of whether the motor windings are energized or de-energized.

Closed Loop - A broadly applied term, relating to any system in which the output is measured and compared to the input. The output is then adjusted to reach the desired condition. In motion control, the term typically describes a system utilizing a velocity and/or position transducer to generate correction signals in relation to desired parameters.

Commutation - The action of applying currents or voltages to the proper motor phases in order to produce optimum motor torque.

Critical Damping - A system is critically damped when the response to a step change in desired velocity or position is achieved in the minimum possible time with little or no overshoot.

DAC - The digital-to-analog converter (DAC) is the electrical interface between the motion controller and the motor amplifier. It converts the digital voltage value computed by the motion controller into an analog voltage. The more DAC bits, the finer the analog voltage resolution. DACs are available in three common sizes: 8, 12, and 16 bit. The bit count partitions the total peak-to-peak output voltage swing into 256, 4096, or 65536 DAC steps, respectively.

Dead Band - A range of input signals for which there is no system response.

Driver - Electronics which convert step and direction inputs to high power currents and voltages to drive a step motor. The step motor driver is analogous to the servo motor amplifier.

Dual Loop Servo – A servo system that combines a velocity mode amplifier/tachometer with a position loop controller/encoder. It is recommended that the encoder not be directly coupled to the motor. The linear scale encoder should be mounted on the external mechanics, as closely coupled as possible to the 'end effector'

Duty Cycle - For a repetitive cycle, the ratio of on time to total time:

Efficiency - The ratio of power output to power input.

Encoder - A type of feedback device which converts mechanical motion into electrical signals to indicate actuator position or velocity.

End Effector – The point of focus of a motion system. The tools with which a motion system will work. Example: The leading edge of the knife is the *end effector* of a three axis (XYZ) system designed to cut patterns from vinyl.

Feed Forward - Defines a specific voltage level output from a motion controller, which in turn commands a velocity mode amplifier to rotate the motor at a specific velocity.

Following Error - The difference between the calculated desired trajectory position and the actual position.

Friction - A resistance to motion caused by contacting surfaces. Friction can be constant with varying speed (Coulomb friction) or proportional to speed (viscous friction).

Holding Torque - Sometimes called static torque, holding torque specifies the maximum external torque that can be applied to a stopped, energized motor without causing the rotor to rotate continuously.

Inertia - The measure of an object's resistance to a change in its current velocity. Inertia is a function of the object's mass and shape.

Kd - K is a generally accepted variable used to represent gain, an arbitrary multiplier, or a constant. The lower case 'd' designates derivative gain.

Ki - K is a generally accepted variable used to represent gain, an arbitrary multiplier, or a constant. The lower case 'i' designates integral gain.

Kp - K is a generally accepted variable used to represent gain, an arbitrary multiplier, or a constant. The lower case 'p' designates proportional gain.

Limits - Motion system sensors (hard limits) or user programmable range (soft limits) that alert the motion controller that the physical end of travel is being approached and that motion should stop.

MCAPI - The Motion Control Application Programming Interface - this is the programming interface used by Windows programmers to control PMC's family of motion control cards.

MCCL - Motion Control Command Language - this is the command language used to program PMC's family of motion control cards.

Open Loop – A control system in which the control output is not referenced or scaled to an external feedback.

Position Error - see following error.

Position Move - Unlike a velocity move, a position move includes a predefined stopping position. The trajectory generator will determine when to begin deceleration in order to ensure the actual stopping point is at the desired target position.

PWM - Pulse Width Modulation is a method of controlling the average current in a motor’s phase windings by varying the duty cycle of transistor switches.

Repeatability - The degree to which the positioning accuracy for a given move performed repetitively can be duplicated.

Resonance - A condition resulting from energizing a motor at a frequency at or close to the motor's natural frequency.

Resolution - The smallest positioning increment that can be achieved.

Resolver - A type of feedback device which converts mechanical position into an electrical signal. A resolver is a variable transformer that divides the impressed AC signal into sine and cosine output signals. The amplitude of these signals represents the absolute position of the resolver shaft.

Slew - That portion of a move made at constant, non-zero velocity.

Step Response - An instantaneous command to a new position. Typically used for tuning a closed loop system, ramping (velocity, acceleration, and deceleration) is not applied nor calculated for the move.

Tachometer - A device attached to a moving shaft that generates a voltage signal directly proportional to rotational speed.

Torque -

Velocity Mode Amplifier – An amplifier that requires a tachometer to provide the feedback used to close the velocity loop within the amplifier.

Velocity Move - A move where no final stopping position is given to the motion controller. When a start command is issued the motor will rotate indefinitely until it is commanded to stop.

Appendix Contents

Appendix

Power Supply Requirements

Part Number	*+5 VDC*	*+12 VDC*	*-12 VDC*	*Unit*
DCX-PCI100	0.9	-----	-----	A
DCX-MC100	.25	.01	.01	A
DCX-MC110	.25	.01 - .5 *	.01	A
DCX-MC400	.25	-----	-----	A
DCX-MC500	.1	*	*	A

* Current depends on output loading

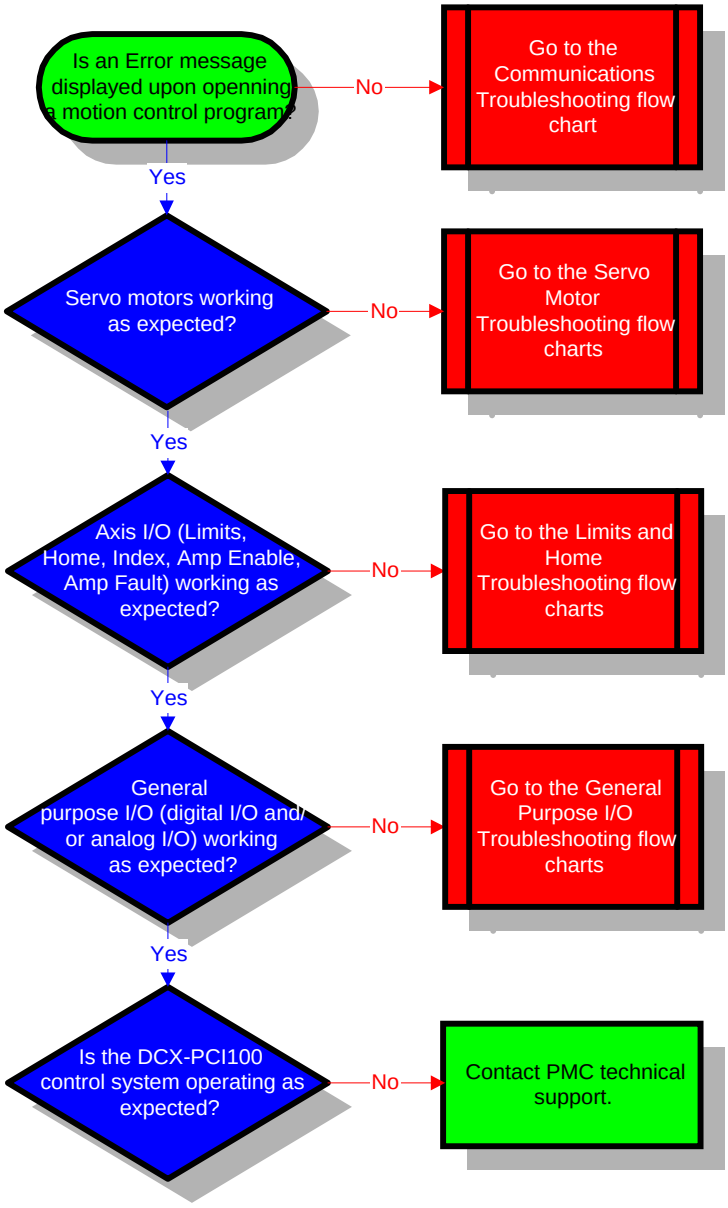
Default Settings

Description	Setting
Programmed Velocity	0
Programmed Acceleration / Deceleration	0
Proportional Gain	0
Derivative Gain	0
Derivative Sampling Frequency	0.000341
Integral Gain	0
Integration Limit	0
Maximum Allowable Following Error	0
Motion Limits	disabled
Low Limit of Movement	0
High Limit of Movement	0
Position Count	0
Optimal Count	0
Index Count	0
Auxiliary Status	0
Position	0
Target	0
Optimal Position	0
Breakpoint Position	0
Position Dead band	0
User Scale	1
User Zero	0
User Offset	0
User Rate Conversion	1
User Output Constant	1

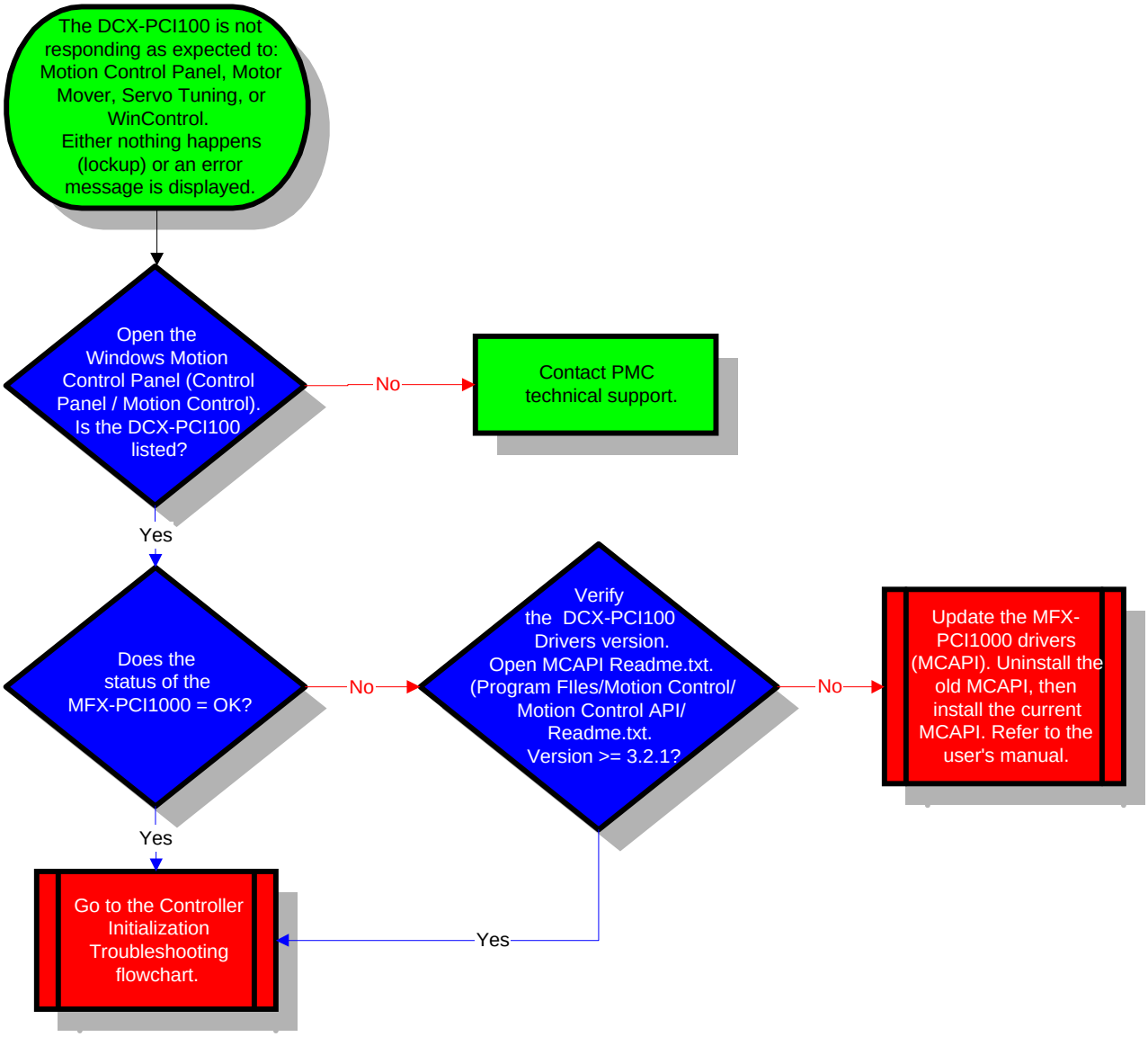
Troubleshooting Controller Operations

On the following pages you will find troubleshooting flow charts to assist the with diagnosis of motion control system failures.

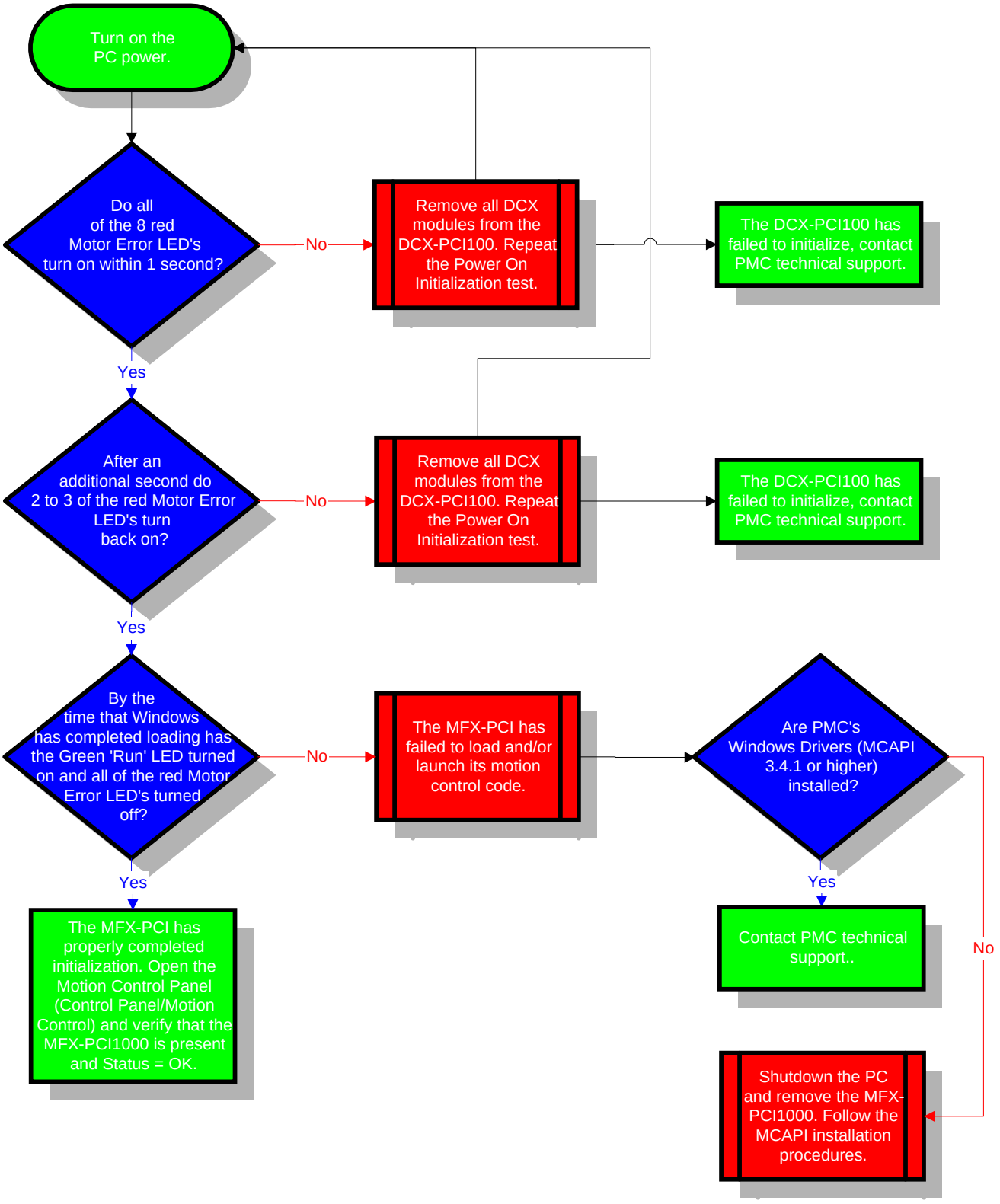
The steps described in these flow charts will direct the user to PMC programs (Motion Integrator, Motor Mover, CWdemo, etc...) and utilities (Servo Tuning, WinControl) that are used to diagnose and resolve system operation.



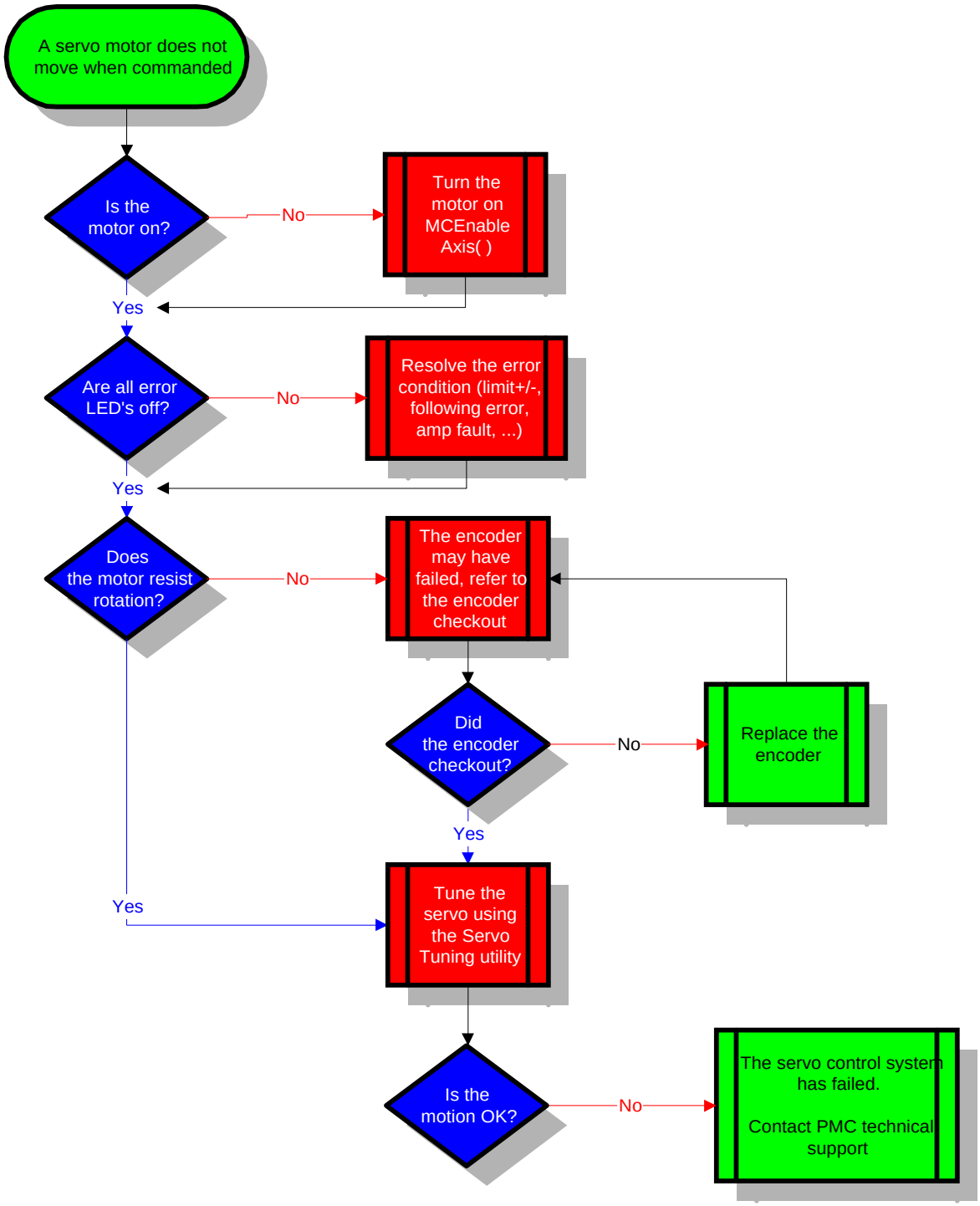
Communications Troubleshooting



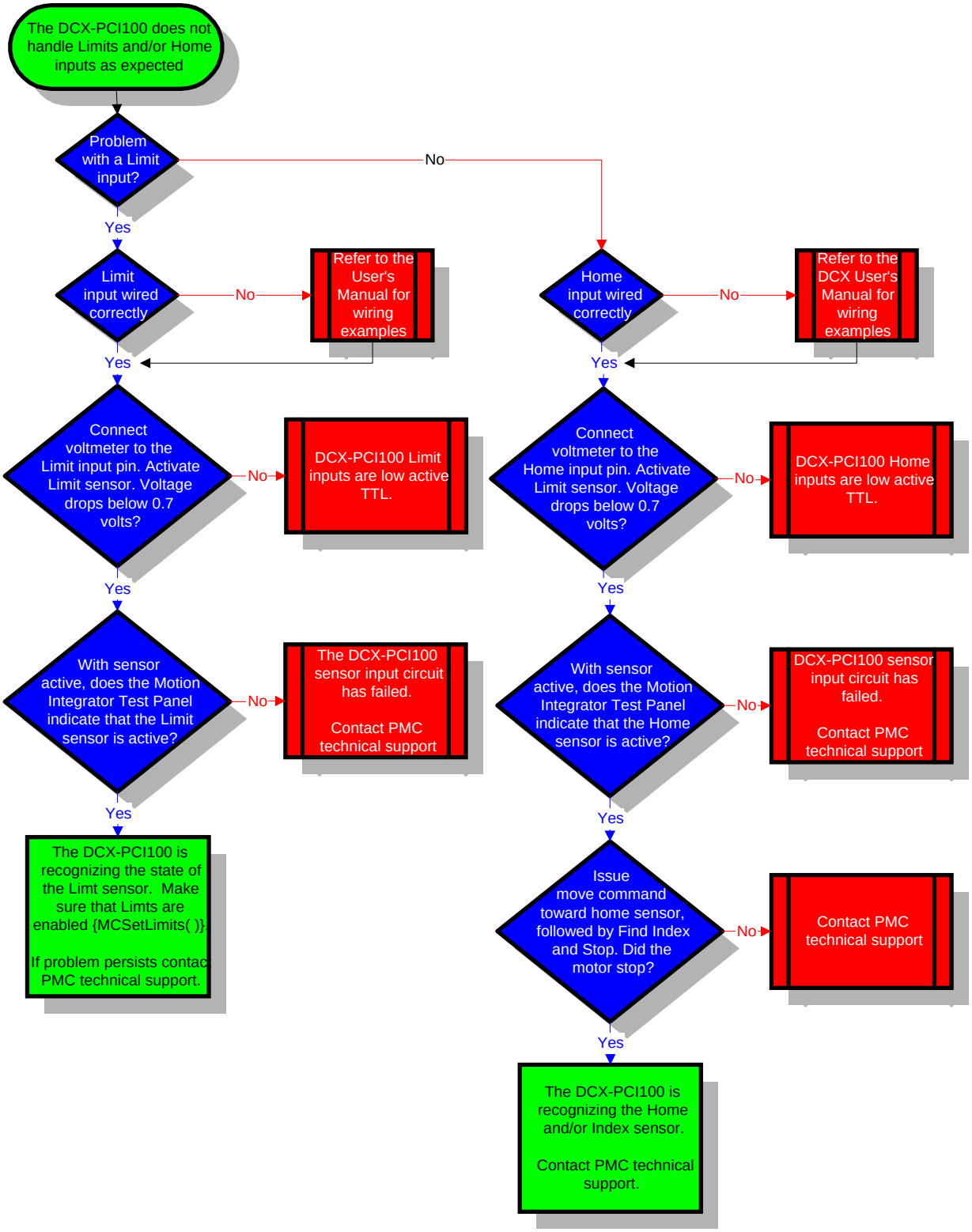
Controller Initialization Troubleshooting



Servo Motion troubleshooting



Limits and Home Troubleshooting



Index

B

C

D

E

F

G

H

I

J

L

M

N

O

P

'

P

Q

R

S

T

U

V

W

Y

Z

Precision MicroControl Corporation
2075-N Corte del Nogal
Carlsbad, CA 92009-1415 USA

Tel: (760) 930-0101
Fax: (760) 930-0222

www.pmccorp.com

Information: info@pmccorp.com
Technical Support: support@pmccorp.com