

# ***DCX-AT200***

*Modular Multi-Axis Motion Control System*

---

## ***Installation and User's Manual***

*Revision 4.0*



---

***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, 1994-2000. 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](http://www.pmccorp.com)

Email:

Information: [info@pmccorp.com](mailto:info@pmccorp.com)

Technical support: [support@pmccorp.com](mailto:support@pmccorp.com)

Sales: [sales@pmccorp.com](mailto:sales@pmccorp.com)

# Table of Contents

Introduction .....	5
Installation.....	11
DCX Motion Control System Installation.....	11
Installing the DCX Software (MCAPI) .....	13
Installing DCX Motor Control and I/O Modules .....	22
DCX-MC200 – Servo Motor Module Installation .....	24
DCX-MC210 – Servo Motor Module Installation .....	28
DCX-MC260 – Stepper Motor Module Installation.....	32
DCX-MC400 – Digital I/O Expansion Module Installation.....	35
DCX-MC500 – Analog I/O Expansion Module Installation.....	36
DCX-MF300 – RS-232 Stand Alone Communication Module Installation.....	36
DCX-MF310 – IEEE-488 Stand Alone Communication Module Installation.....	37
Programming, Software and Utilities .....	41
Controller Interface Types.....	42
Building Application Programs using Motion Control API.....	43
PMC Sample Programs .....	48
Motion Integrator .....	49
PMC Utilities.....	52
MCAPI On-line Help.....	55
Communication Interfaces .....	59
PC communications Interfaces .....	60
RS-232 Communications Interface .....	60
IEEE-488 Communications Interface.....	61
DCX Operation Basics.....	63
Introduction .....	63
Low Level DCX Operations.....	64
Motion Control .....	69
Theory of DCX Motion Control.....	69
DCX Servo Basics.....	70
Tuning the Servo.....	74
DCX Stepper Basics .....	87
Closed Loop Steppers.....	88
Moving Motors with PMC demo's .....	92
Defining the Characteristics of a Move .....	93
Velocity Profiles.....	94
Point to Point Motion .....	95
Constant Velocity Motion .....	95
Contour Motion (arcs and lines).....	96
Electronic Gearing.....	104
Jogging.....	105
Defining Motion Limits.....	107
Homing Axes.....	109
Motion Complete Indicators .....	118
On the Fly changes .....	119
Feed Forward (Velocity, Acceleration, Deceleration) .....	120
Save and Restore Axis Configuration .....	121
Application Solutions .....	123
Auxiliary Encoders .....	123
Backlash Compensation .....	127
Emergency Stop.....	128
Encoder Rollover.....	130
Flash Memory Firmware Upgrade .....	131
Laser Cutting.....	132
Learning/Teaching Points .....	135
Record Motion Data .....	136

## Table of Contents

Manually Resetting the DCX .....	137
Tangential Knife Control .....	138
Threading Operations .....	140
Torque Mode Output Control .....	141
Defining User Units .....	143
DCX Watchdog .....	145
General Purpose I/O .....	149
DCX Motherboard Digital I/O .....	149
Configuring the DCX Digital I/O .....	150
Using the DCX Digital I/O .....	152
DCX Motherboard Analog Inputs .....	154
DCX Module Analog I/O .....	155
Using the Analog I/O .....	156
Calibrating the MC500/MC520 +/- 10V Analog Outputs: .....	158
Motion Control API Function Reference .....	161
Motion Control API Function Quick Reference Tables .....	163
Setup Commands .....	166
Motion Functions .....	178
Reporting Functions .....	193
I/O Functions .....	210
Macros and Multi-Tasking .....	214
MCAPI Driver Functions .....	216
DCX Specifications .....	223
Motherboard: DCX-AT200 .....	223
DCX-MC200 - +/- 10 Volt Analog Servo Motor Control Module .....	224
DCX-MC210 - PWM Motor Drive Servo Control Module .....	225
DCX-MC260 - Stepper Motor Control Module .....	226
DCX-MC400 - 16 channel Digital I/O Module .....	227
DCX-MC5X0 - Analog I/O Module .....	227
DCX-MF300 - RS-232 Communications Interface Module .....	229
DCX-MF310 - IEEE-488 Communications Interface Module .....	229
Connectors, Jumpers, and Schematics .....	231
DCX-AT200 Motion Control Motherboard .....	231
DCX-MC200 +/- 10V Servo Motor Control Module .....	238
DCX-MC210 PWM Motor Drive Servo Control Module .....	243
DCX-MC260 Stepper Motor Control Module .....	249
DCX-MC400 Digital I/O Module .....	254
DCX-MC500/510/520 Analog I/O Module .....	256
DCX-MF300 – RS-232 Interface Module .....	258
DCX-MF310 IEEE-488 Interface Module .....	262
DCX-BF022 Relay Rack Interface .....	266
DCX-BF100 Servo Module Interconnect Board .....	270
DCX-BF160 Stepper Module Interconnect Board .....	275
DCX MCCL Commands .....	281
Introduction to MCCL (low level command set) .....	281
MCCL Command Quick Reference Tables .....	283
Building MCCL Macro Sequences .....	286
MCCL Multi-Tasking .....	288
Downloading MCCL Text Files .....	290
Single Stepping MCCL Programs .....	290
Outputting Formatted Message Strings .....	292
PLC I/O Control using MCCL Sequence Commands .....	292
PLC Control and DCX Analog I/O .....	296
DCX User Registers .....	298
Reading Data from DCX Memory .....	299
DCX Scratch Pad Memory .....	302
MCCL Command Set Description .....	303
Setup Commands .....	303

Mode Commands .....	315
Motion Commands .....	317
Reporting Commands .....	326
I/O Commands .....	334
Macro and Multi-Tasking Commands.....	337
Register Commands.....	339
Sequence (If/Then) Commands .....	345
Miscellaneous Commands .....	351
Troubleshooting .....	355
Controller Error Codes.....	365
MCAPI Error Codes .....	366
MCCL Error Codes.....	367
Printing a PDF Document.....	369
Glossary.....	371
Appendix.....	377
Dual Ported Memory .....	377
Binary Communication Interface.....	382
ASCII Communication Interface.....	386
Power Supply Requirements .....	387
Default Settings.....	388
Pause and Resume Motion.....	389
Physical Assignment of Axes Numbers .....	389
Multiple User Communications Interfaces .....	390
Stand Alone Applications .....	391
On-Board File Operations .....	392
HPGL Plotting .....	395
Index .....	401

### User manual revision history

<b>Revision</b>	<b>Date</b>	<b>Description</b>
1.0a	9/1/94	Initial release
1.1a	11/1/94	Added cubic spline interpolation to contouring. Added Profile Parabolic command. Added Move Absolute command description. Made miscellaneous corrections and changes.
1.5a	3/20/96	Added description of backlash compensation function and commands. Added description of file commands. Added description of plotting commands. Added appendix describing HPGL commands. Created new page layout with standardized headers and footers. Changed format of command descriptions. Made miscellaneous corrections and changes.
4.0	2/1/2K	Added firmware revisions to rev. <b>4.0a</b> Added MCAPI revisions to rev <b>2.20.0000</b> Added references to Motion Integrator <b>1.0a</b> Added Flash Wizard text (requires firmware 3.6a or higher) Converted to Word - User Manual Format Changed organization structure Added Application Solutions chapter Added Troubleshooting chapter Converted all command references from MCCL to MCAPI Added descriptions of Feed forward and Velocity Mode Amplifier servo tuning Added C++, Visual Basic, Delphi, and LabVIEW program build information

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](http://www.pmccorp.com)

Email:

Information: [info@pmccorp.com](mailto:info@pmccorp.com)

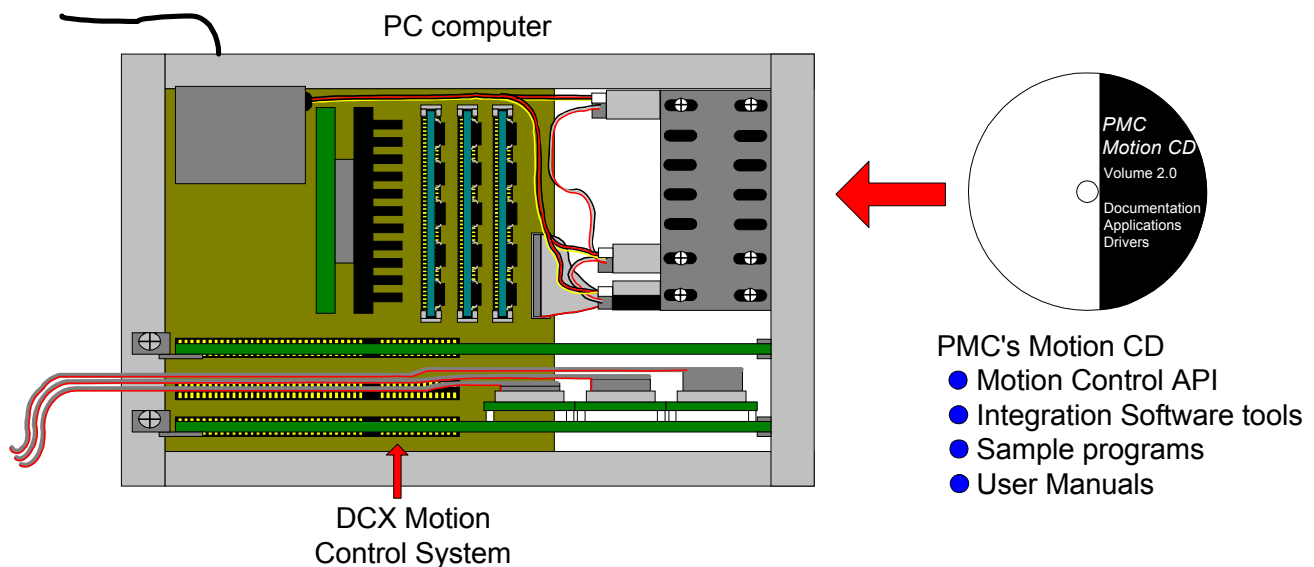
Technical support: [support@pmccorp.com](mailto:support@pmccorp.com)

Sales: [sales@pmccorp.com](mailto:sales@pmccorp.com)

## Introduction

---

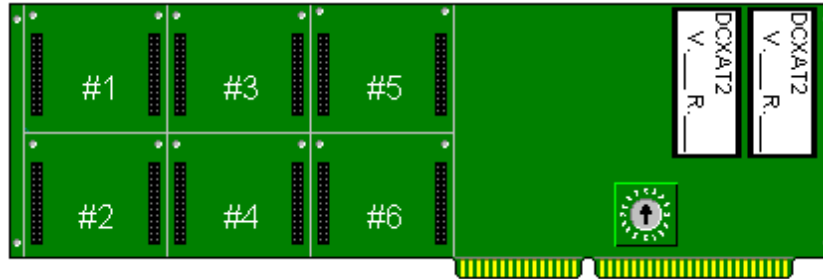
This document describes the installation and use of the DCX-AT200 Modular Multi-Axis Motion Control System. In a typical application, the DCX is installed in an Intel compatible PC computer. The PC will run high level language (C++, Visual Basic, Delphi, LabVIEW) application programs written by the machine builder, that uses Precision MicroControl's Motion Control API to issue high level motion functions to one or more DCX controllers. The DCX executes these motion functions independent of the host. The DCX Motion Control system can be installed in most any Windows PC computer that has available ISA slots. The DCX requires no minimum system hardware, memory size, or configuration.



The modular architecture of the DCX system allows the user to '**mix and match**' components to meet the specific requirements of each application. The DCX system controls the motion of any combination of servos and stepper motors simultaneously. In addition the DCX system supports; direct motor drive (no servo amplifier required) of small servo motors, expandable digital and analog I/O, and stand alone applications.

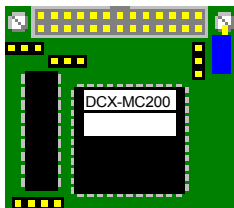
The term DCX refers to a system consisting of from 1 to 7 circuit boards assembled together to form a motion control assembly. The main component of the assembly is the DCX-AT200 "motherboard". It is a 'full' size (approximately 5" x 13") ISA peripheral card.

The DCX-AT200 motherboard is the platform upon which the DCX motion control system is built. It communicates with the PC host via the ISA bus. On board dual ported memory is used to pass data between the DCX-AT200 and the PC. The on board CPU (Intel I960) allows the DCX-AT200 to operate autonomously from the PC, freeing the host to process critical events while the DCX handles all motion control. The DCX-AT200 motherboard also includes 16 general purpose TTL level digital I/O channels and 4 analog input channels (8 bit, 0 to +5V).



On this DCX-AT200 motherboard, the user can install as many as 6 smaller "daughter boards" known as "DCX modules". There are seven DCX module types available which allow the DCX system to be configured for a variety of applications. 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-MC200 - Servo Motor Control Module

+/- 10 volt, 12 bit analog command output for use with a servo amplifier/drive

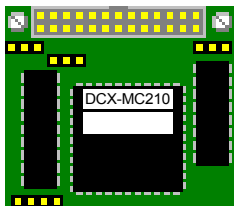
Inputs - Encoder Coarse Home, Limit +, and Limit -, Amplifier Fault

Output - Amplifier Enable

Quadrature Incremental Encoder Interface

Primary - Single ended (A, B, Z) or Differential (A+, A-, B+, B-, Z+, Z-)

Auxiliary - Single ended (A, B, Z+, Z-)



#### DCX-MC210 - Servo Motor Control Module

PWM output for direct drive of small motors (12W, 12 volt @ 1 A)

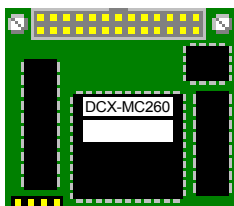
Inputs - Encoder Coarse Home, Limit +, and Limit -, Amplifier Fault

Output - Amplifier Enable

Quadrature Incremental Encoder Interface

Primary - Single ended (A, B, Z) or Differential (A+, A-, B+, B-, Z+, Z-)

Auxiliary - Single ended (A, B, Z+, Z-)



#### DCX-MC260 - Stepper Motor Control Module

Pulse/CCW and Direction/CW outputs for use with a stepper driver

Inputs - Home, Limit +, and Limit -, Null

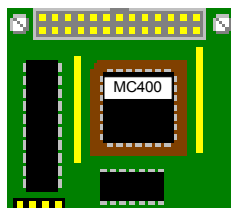
Outputs - Drive Enable, Half/Full step, Full/Half current

Quadrature Incremental Encoder Interface

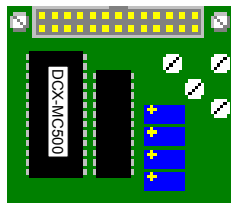
Auxiliary - Single ended (A, B, Z) or Differential (A+, A-, B+, B-, Z+, Z-)



## DCX General Purpose I/O Modules



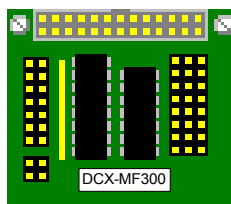
DCX-MC400 – 16 Channel Digital I/O Expansion module  
Each channel is individually programmable as either an input or output  
TTL level (0 – 5 volt, 2 ma sink/source)



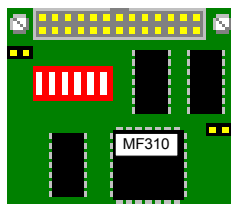
DCX-MC500 – Analog I/O Expansion module  
Inputs – 4 channels, 0 – 5 volts, 12 bit  
Outputs – 4 channels, 0 – 5 volts and/or –10 - +10 volts, 12 bit

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

## DCX Auxiliary Communication Modules for Stand Alone Applications

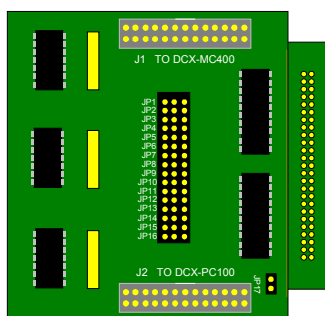


DCX-MF300 – RS232 Interface Module  
ASCII command interface for stand alone applications



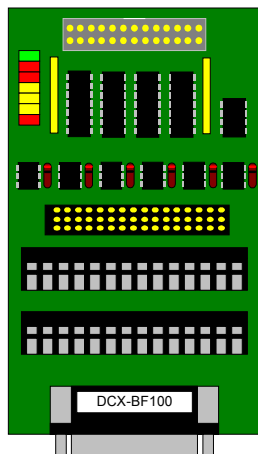
DCX-MF310 – IEEE-488 Interface Module  
ASCII command interface for stand alone applications

## DCX Interconnect and Isolation Assemblies



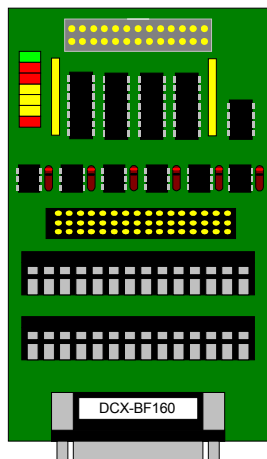
DCX-BF022 – Opto 22 Relay Rack Interface  
16 digital I/O channels, each channel individually configured via user installed jumper  
as input or output. Connects to DCX digital I/O via 26 conductor ribbon cable.

## DCX Interconnect and Isolation Assemblies (continued)



DCX-BF100 – Opto isolation and Interconnect assembly for DCX Servo Motor Control Modules (DCX-MC200, DCX-MC210)  
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-BF160 – Opto isolation and Interconnect assembly for DCX Stepper Motor Control Module (DCX-MC260)  
Opto isolated inputs – Home, Limit +, Limit -, Null, Enc. Coarse Home  
Open collector output – Drive Enable  
Differential receiver for Index +, Index –  
External system connections via DB25 or two 14 contact screw terminal strip

LED indicators for:  
Drive enable  
Home  
Encoder Coarse Home  
Limit +  
Limit –  
Null Position



## **Chapter Contents**

---

- PC based Application Installation
- DCX-AT200 Controller Installation
- Installing the DCX Software (MCAPI)
- Installing DCX Motor Control and I/O Modules
- DCX-MC200 – Servo Motor (+/- 10V output) Module Installation
- DCX-MC210 – Servo Motor (PWM output for direct motor drive) for Module Installation
- DCX-MC260 – Stepper Motor Module Installation
- DCX-MC400 – Digital I/O Expansion Module Installation
- DCX-MC500 – Analog I/O Expansion Module Installation
- DCX-MF300 – RS-232 Stand Alone Communication Module Installation
- DCX-MF310 – IEEE-488 Stand alone Communication Module Installation

## Installation

---

Typically the DCX is installed in an ISA slot of a PC computer or the passive back plane of an industrial computer. Power (+5V, +12V, and –12V), Ground reference, and communications (Address, Data, and Read/Write control signals) are supplied via the ISA edge connector of the DCX.

The DCX on-board intelligence enables it to operate as a stand-alone controller. Optional RS-232 or IEEE-4888 communication interfaces allow a host computer to communicate with the DCX for programming and/or operator control. For stand-alone installation information please refer to **Stand-alone Application** section in the **Appendix** at the end of this user manual.

## DCX Motion Control System Installation

The basic steps of installing a DCX motion controller for PC based applications are as follows:

- Select the memory address of the DCX
- Install the DCX in an available ISA slot
- Install the DCX software (MCAPI, drivers and utilities)
- Install and wire DCX servo and stepper motion control modules
- Install and wire digital and analog I/O modules
- Verify servo operation (refer to **DCX Servo Basics** in the **Motion Control Chapter**)
- Verify stepper operation (refer to **DCX Stepper Basics** in the **Motion Control Chapter**)
- Verify I/O operation (refer to **DCX General Purpose I/O Chapter**)

### Setting the memory address of the DCX

When installed in a PC, the dual ported memory of the DCX occupies 4K of the PC's memory space. As shipped from the factory, the DCX is configured to reside at memory address D000H. This configuration is defined by the settings of jumper JP8 (base memory address) and SW1 (memory address offset). The addressing of the DCX should be left as is unless it is determined that another device resides at the same location.

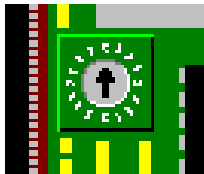
In most Intel compatible PC's, the 64K area from D000 hex to DFFF hex is available and has been chosen as the factory default memory range for the DCX. Jumper JP8 supports memory address ranges from 8000 hex to F000 hex. The following table details the possible settings of jumper JP8.

### JP8 - IBM-PC Interface base memory address select

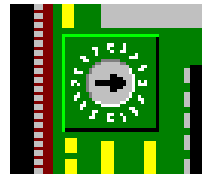
<b>Base address</b>	<b>JP8 5 to 6</b>	<b>JP8 3 to 4</b>	<b>JP8 1 to 2</b>
8000 hex	connected	connected	connected
9000 hex	connected	connected	open
A000 hex	connected	open	connected
B000 hex	connected	open	open
C000 hex	open	connected	connected
<b>D000 hex</b>	<b>open</b>	<b>connected</b>	<b>open</b>
E000 hex	open	open	connected
F000 hex	open	open	open

Most other ranges that can be set with jumper JP8 will result in hardware conflicts within the host computer when the DCX is installed. Unless the user determines that another range is required, jumper JP8 should be left at its default setting as shown in appendix B.

The 16 position rotary SW1 is used to define at which 4K offset the DCX will reside. With jumper JP8 set to the factory defined configuration the DCX will occupy 4K bytes somewhere between D000 hex and DFFF hex in the PC's memory space as selected by the rotary switch. If the rotary switch SW1 on the motherboard is set to 0, the DCX will occupy D000:0000 hex through D000:0FFF hex. If it is set to 1, it will occupy D000:1000 hex through D000:1FFF hex, and so on. In the host's memory map, the lowest numbered memory location that a DCX board occupies is referred to as the 'base' address. The two graphics below show the memory switch setting for a DCX at address D0000:0000 (SW1=0) and D000:4000 (SW1=4).



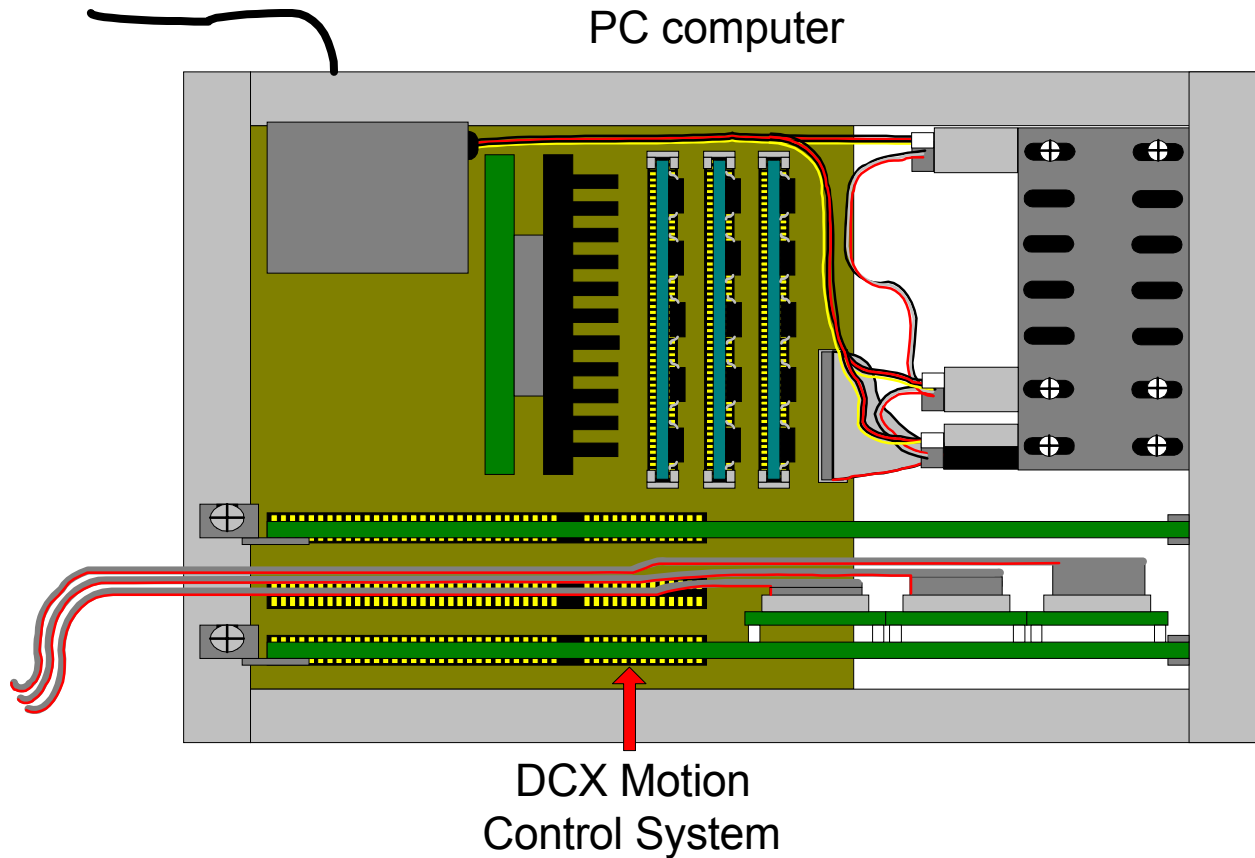
Default memory address switch SW1=0  
D000:0000 - D000:0FFFH



Memory address switch SW1=4  
D000:4000 - D000:4FFFH

### Install the DCX in an available ISA slot

ISA slots are memory address independent, the DCX can be installed in any of the PC's available ISA with no changes in jumpering. The DCX modules and cabling may interfere with an ISA 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.



## Installing the DCX Software (MCAPI)



DCX controllers ship with PMC's Motion CD 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](http://www.pmccorp.com)

### Installation from PMC's Motion CD

To install the Motion Control API software which includes: setup, integration, and diagnostic utilities, 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.



The Motion Control API will can be installed 'on top of' previous installations, there is no need to remove earlier versions of the MCAPI. Starting with version 2.20 the MCAPI default directory was changed to /Program Files/Motion Control/Motion Control API. Previous versions of the MCAPI were installed in /PMC/MCAPI.

## Installation

The following windows should be displayed:



Step #1 - Select "Support"



Step #2 - Select "DCX-AT200"



Step 3) Select "Windows"



Step #4) Choose the appropriate installation. Follow the on screen instructions.

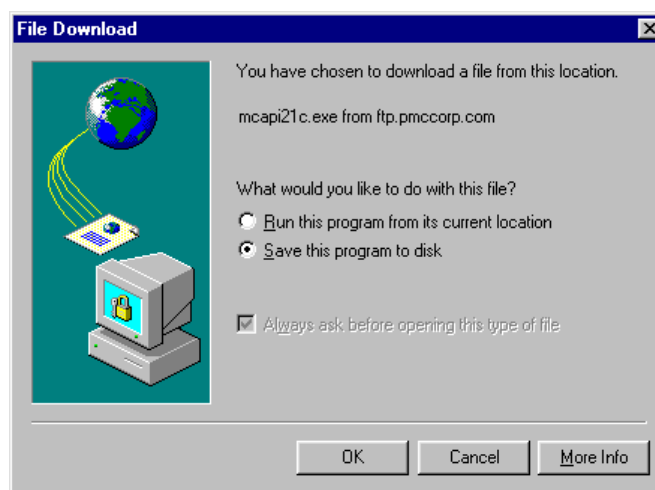
### Downloading the Most Recent release of the Motion Control API from PMC's web site

Due to the dated nature of a CD, it is recommended that the user check PMC's web ([www.pmccorp.com](http://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.

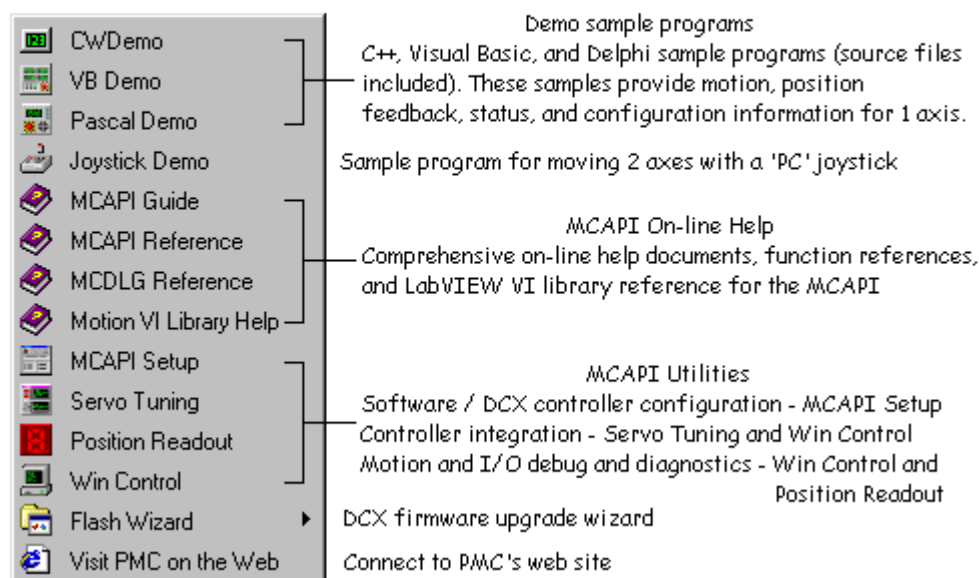


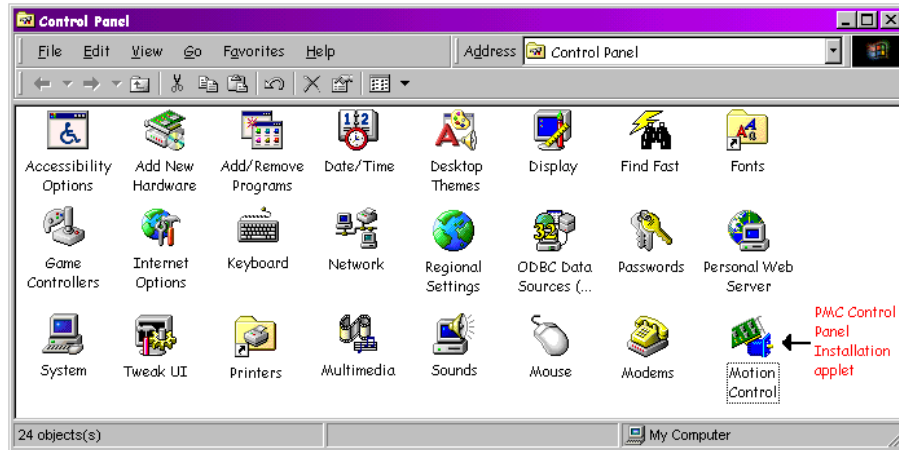
The installation of the MCAPI will begin upon launching the downloaded file. Follow the on screen instructions.

### 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. For additional information on individual MCAPI components please refer to the **Software** and **Utilities** section of the **Programming, Software, and Utilities** chapter of this manual.

## PMC MCAPI Components



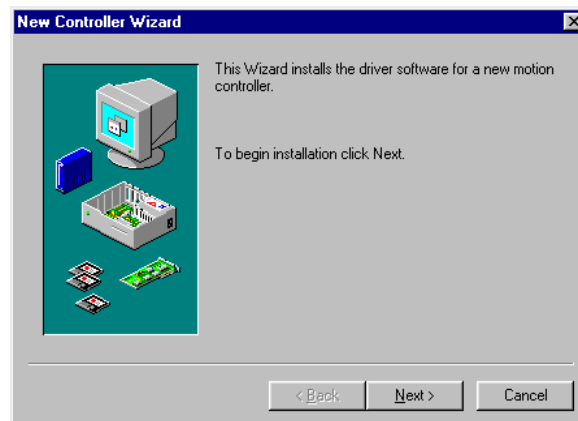


### Configure the MCAPI driver with the Setup applet

The MCAPI device driver must be configured for the type, memory address, and quantity of DCX controllers installed in your computer. 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).



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



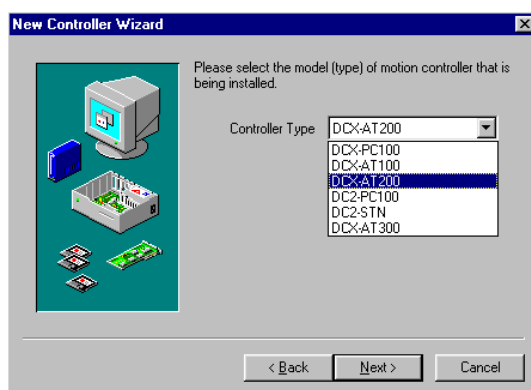
### Controller ID

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 selected to match the rotary switch setting (memory offset) of the DCX controller. This simplifies keeping track of memory addresses, etc.



### Controller Type

The MCAPI supports mixing and matching various PMC controllers (DCX-AT300, 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-AT200.



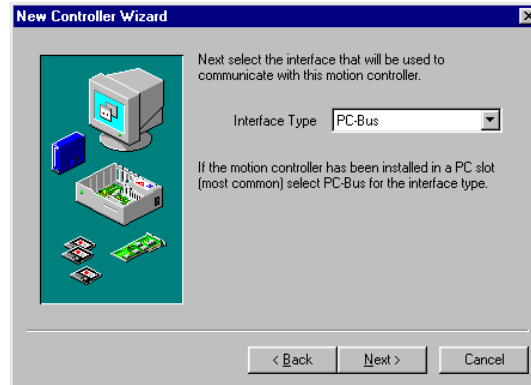
### Description

Allows the user to enter comments about the controller. An example of a completed General setup of a DCX-AT200 follows:



## Communications Interface

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

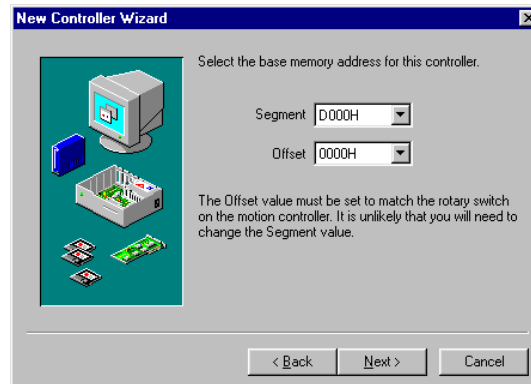


## Define the Memory Address of the DCX Controller

Select the Interface tab to configure the MCAPI for the address of the Dual Ported Memory of the DCX. For additional information about the memory address of the DCX please refer to the description of **Setting the memory address of the DCX** earlier in this chapter.



Although the hardware of the DCX controllers supports memory addresses as low as 8000:0000, under Windows the lowest address range available for the controller is A000:0000. The DCX driver will not allow you to select an address any lower than that.

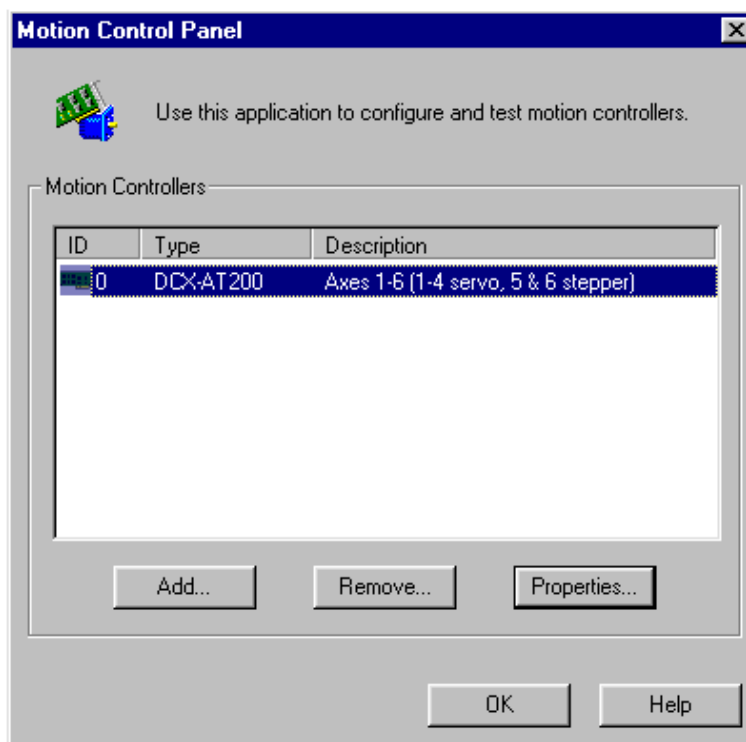


## Testing the Installation

The final step of the New Controller Wizard is to test the communication between the Motion Control API and the DCX motion controller.

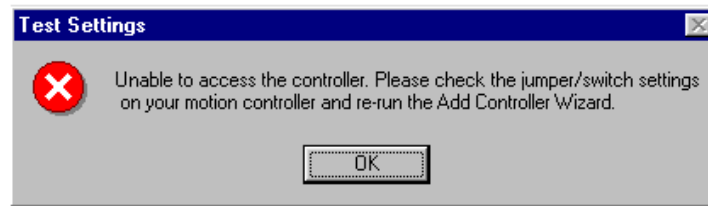


A successful controller communication test will result in the following displays:



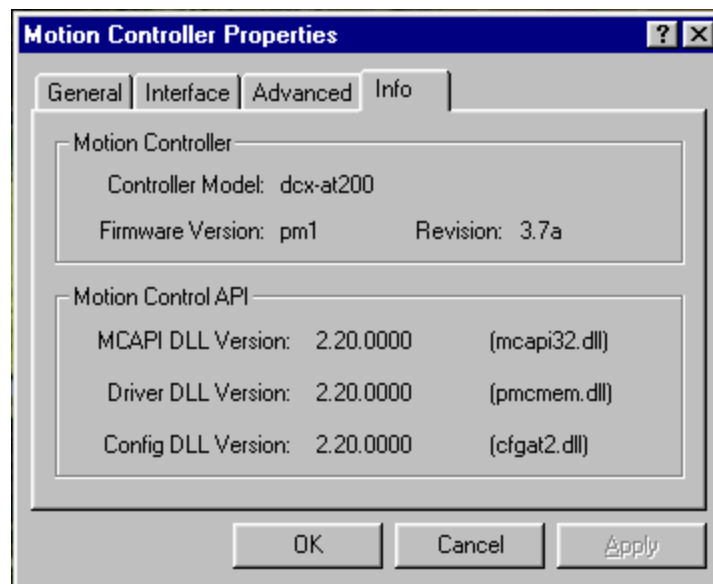
To complete the New Controller Wizard select OK. If you need to configure another controller select Add.

If the test fails and the New Controller Wizard is unable to verify communication between the controller and the Motion Control API refer to the troubleshooting guide in this manual.



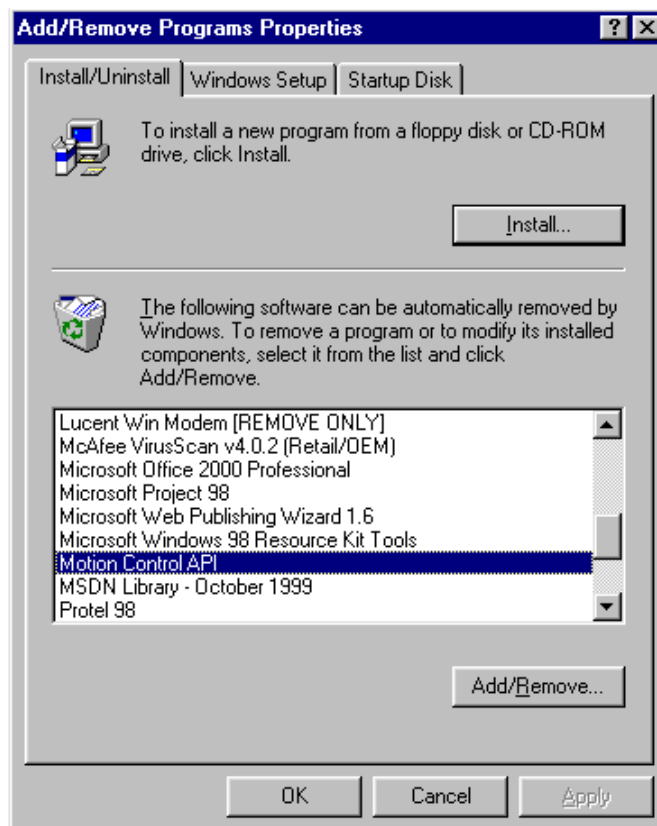
### Reporting Software and Firmware Versions

From the Motion Control panel you can view the installed versions of the Motion Control API and the on-board firmware of the DCX-AT200 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.



### Removing the Motion Control API

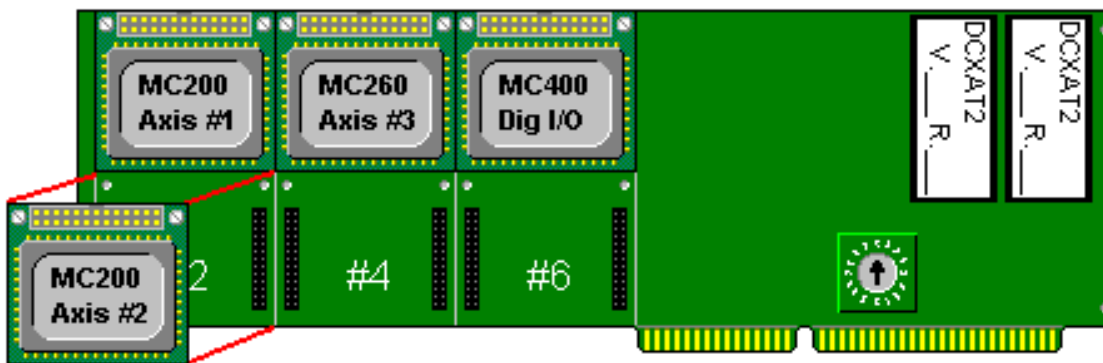
To remove the MCAPAPI, launch the Add/Remove Programs applet in the Windows Control Panel. After the Uninstall Shield has removed the MCAPAPI you will need to restart the computer to remove active .dll's. If you are going to install an older version of the MCAPAPI (2.1c or earlier) you will need to manually delete the file mcapi.ini from the Windows folder.



## 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 six 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.



If a group of motors will be required to perform multi-axis contouring motion, one axis of the group should be assigned to axis 1. This will be the controlling axis for the group. Other groups of axes on the controller can also perform contouring motion, but will be more limited in the number of motion segments that can be stored on the board. For additional information on multi-axis contouring please see the description of **Contouring Motion (arcs and lines)** in the **Motion Control** chapter of this manual.

To install the modules, lay the DCX-AT200 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.

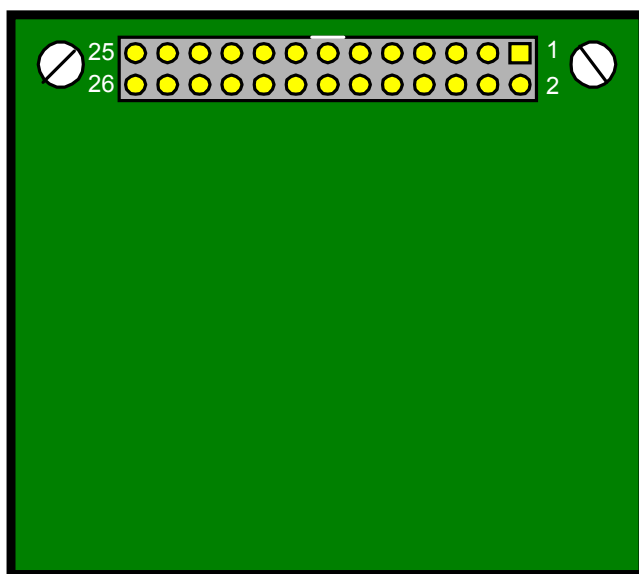
Next the DCX should be re-installed in the PC chassis and interfacing cables connected. Refer to the following sections in this chapter for specific jumper and wiring information for the types of modules that are being used. When cabling has been completed, power can be applied to the system and initial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





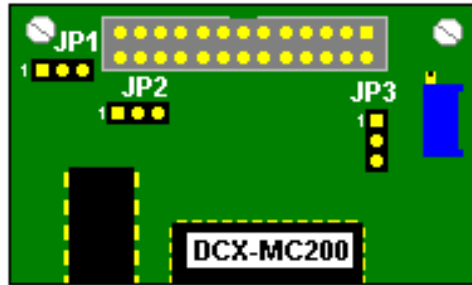
Please note that all DCX modules contain a 26 pin, shrouded, center polarized header for I/O connections. The pins of this connector are numbered from 1 to 26. The following diagram shows the location of pins 1, 2, 25 and 26. The other 22 pins are numbered and located respectively.

## DCX MODULE CONNECTOR PIN NUMBERING (TOP SIDE VIEW)



## DCX-MC200 – Servo Motor Module Installation

Installation of a DCX-MC200 Servo Motor Control Module includes setting three jumpers (JP1, JP2, and JP3). These jumpers are used to configure the module for the type of incremental encoder that will be used. These jumpers are configured by installing shorting blocks on the pins of the jumpers, or leaving them open. Note that the pins of the three jumpers are numbered sequentially from 1 to 3, with pin 1 being shown as a square.



Jumper JP1 configures the module's encoder phase inputs for 'single ended' (A and B) or 'differential' (A+, A-, B+, and B-) signals. For single ended outputs, install the 3 hole shorting block (supplied with the module) across all 3 pins of jumper JP1. Connect the A and B signals from the encoder to the A+ and B+ inputs of the module. If an encoder with differential phase outputs is to be connected to the module, jumper JP1 should be left open (no shorting block installed).

Jumper JP2 is used to configure the module's encoder index input. Either a single ended or differential index signal can be connected to the module. The table below lists the possible combinations.

MC200 Module Encoder Index Configuration

Signal Name	Input Type	Active Level	Jumper JP2	J3 connections
Index +	Single ended	High	1 to 2	Pin 8
Index -	Single ended	Low	2 to 3	Pin 25
Index +/-	Differential	N/A	None	Pins 8(+) & 25(-)

The MC200 provides a user selectable Encoder Power supply by routing +5V or +12V from the PC supply to connector J3 pin 17. The Encoder Power supply can provide up to 500ma of current.

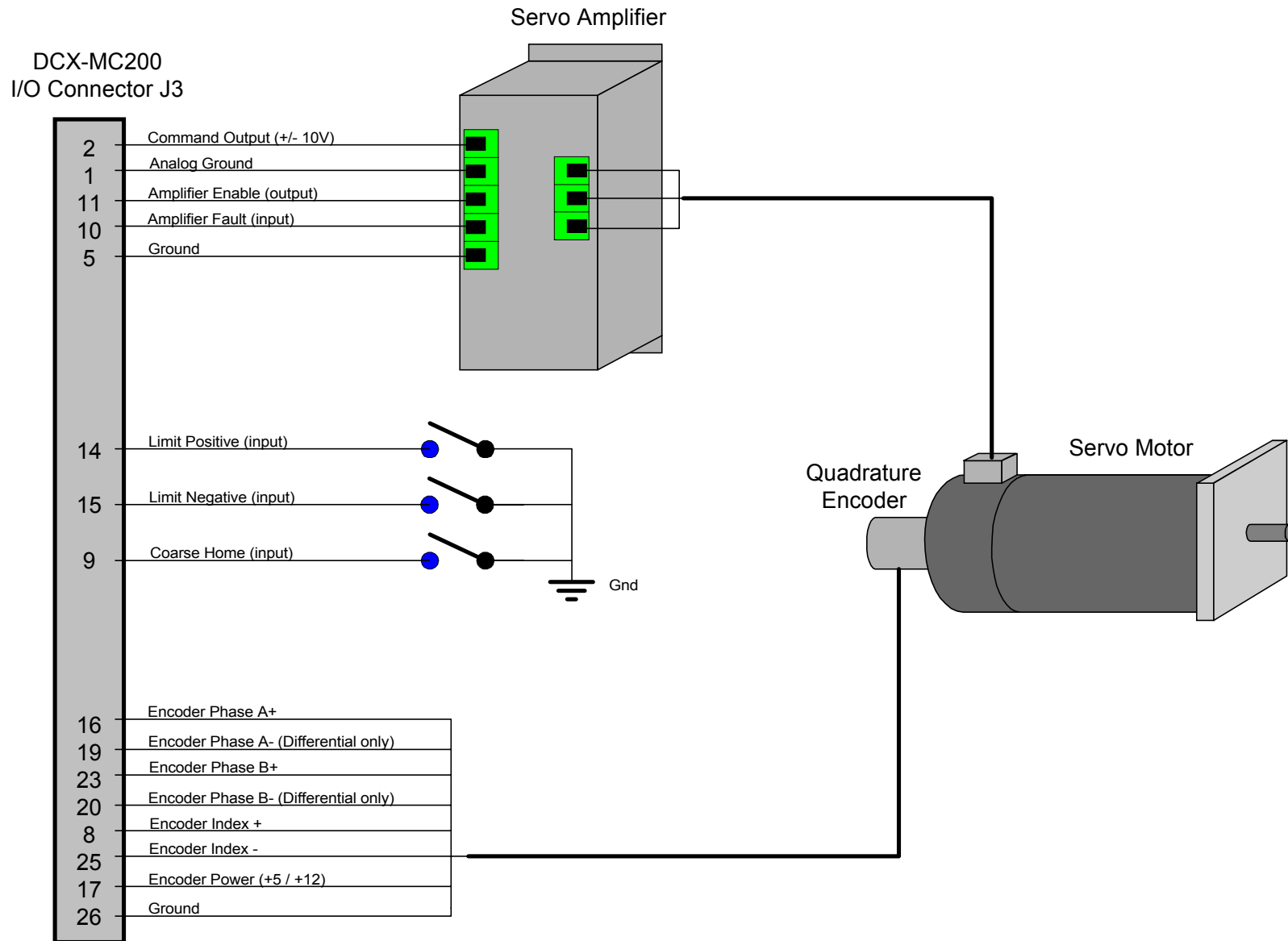
MC200 Encoder Power Selection Jumper

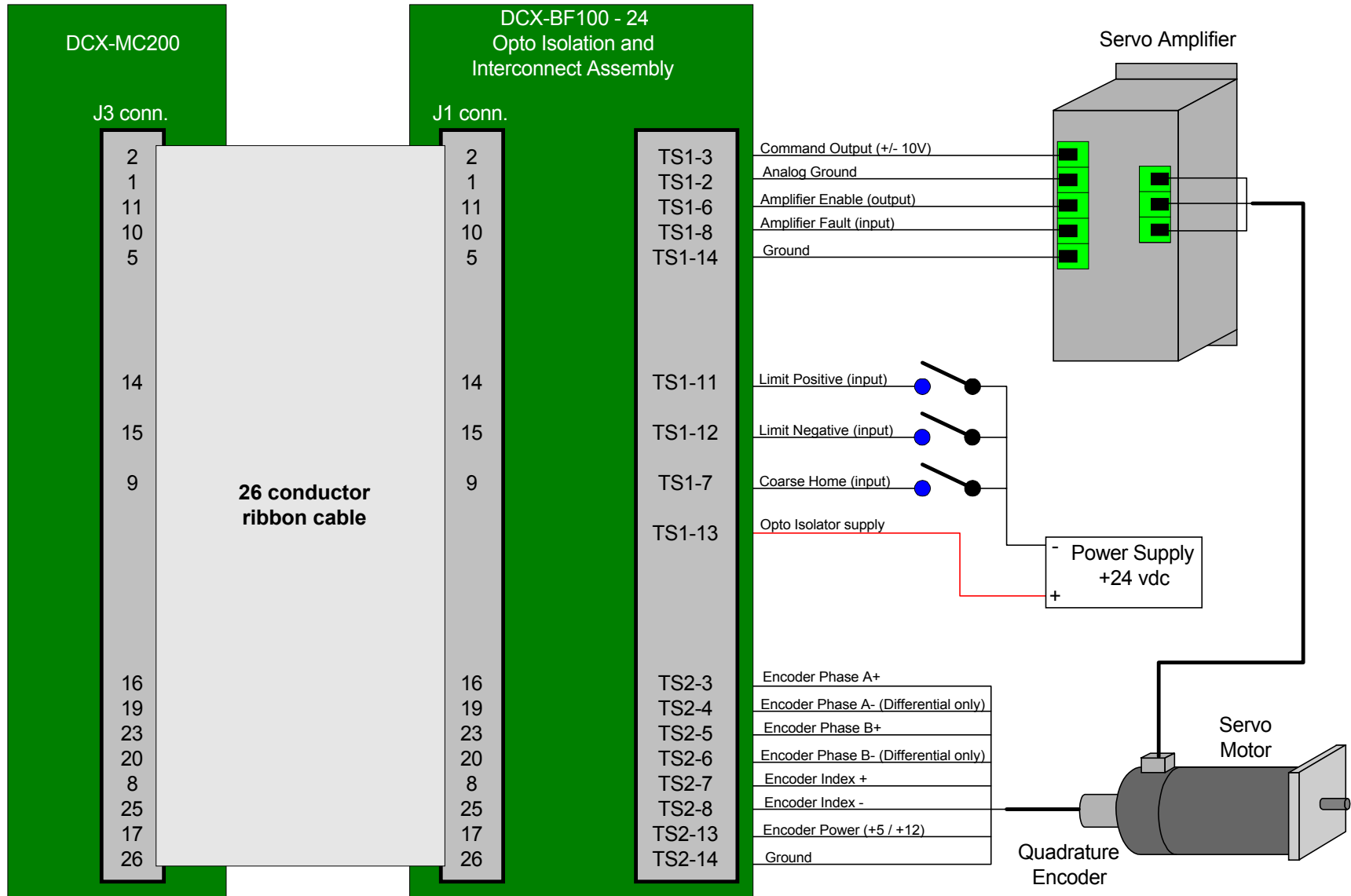
Jumper	Setting	Encoder Power Supply (JP3 pin 17)
JP3	Pins 1 to 2	+5VDC
JP3	Pins 2 to 3	+12 VDC
JP3	Open	Open



**Note:** The DCX-MC200 provides the Encoder Power output as a convenience, It is **not required** that it be used to power the encoder. If an external +5 volts or +12 volts supply is used to power the encoder, jumper JP3 **must still be configured** to match the voltage level (+5 volts or +12 volts) of the external supply.

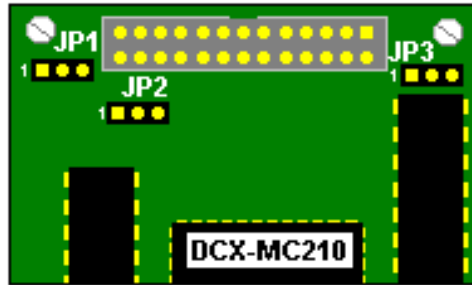
After configuring the jumpers of the module, the servo encoder, amplifier and limit switches can be connected to the module. Wiring diagrams on the next two pages depict typical installations. The first diagram details direct connection of the MC200 to the external components (servo amplifier, encoder, and sensors). The second diagram details typical connections when a **DCX-BF100 Opto Isolation and Interconnect Assembly** is used.





## DCX-MC210 – Servo Motor Module Installation

Installation of a DCX-MC210 Servo Motor Control Module includes setting three jumpers (JP1, JP2, and JP3). These jumpers are used to configure the module for the type of incremental encoder that will be used. These jumpers are configured by installing shorting blocks on the pins of the jumpers, or leaving them open. Note that the pins of the three jumpers are numbered sequentially from 1 to 3, with pin 1 being shown as a square.



Jumper JP1 configures the module's encoder phase inputs for 'single ended' (A and B) or 'differential' (A+, A-, B+, and B-) signals. For single ended outputs, install the 3 hole shorting block (supplied with the module) across all 3 pins of jumper JP1. Connect the A and B signals from the encoder to the A+ and B+ inputs of the module. If an encoder with differential phase outputs is to be connected to the module, jumper JP1 should be left open (no shorting block installed).

Jumper JP2 is used to configure the module's encoder index input. Either a single ended or differential index signal can be connected to the module. The table below lists the possible combinations.

Table 2: MC210 Module Encoder Index Configuration

Signal Name	Input Type	Active Level	Jumper JP2	J3 connections
Index +	Single ended	High	1 to 2	Pin 8
Index -	Single ended	Low	2 to 3	Pin 25
Index +/-	Differential	N/A	None	Pins 8(+) & 25(-)

The MC210 provides a user selectable Encoder Power supply by routing +5V or +12V from the PC supply to connector J3 pin 17. The Encoder Power supply can provide up to 500ma of current.

MC200 Encoder Power Selection Jumper

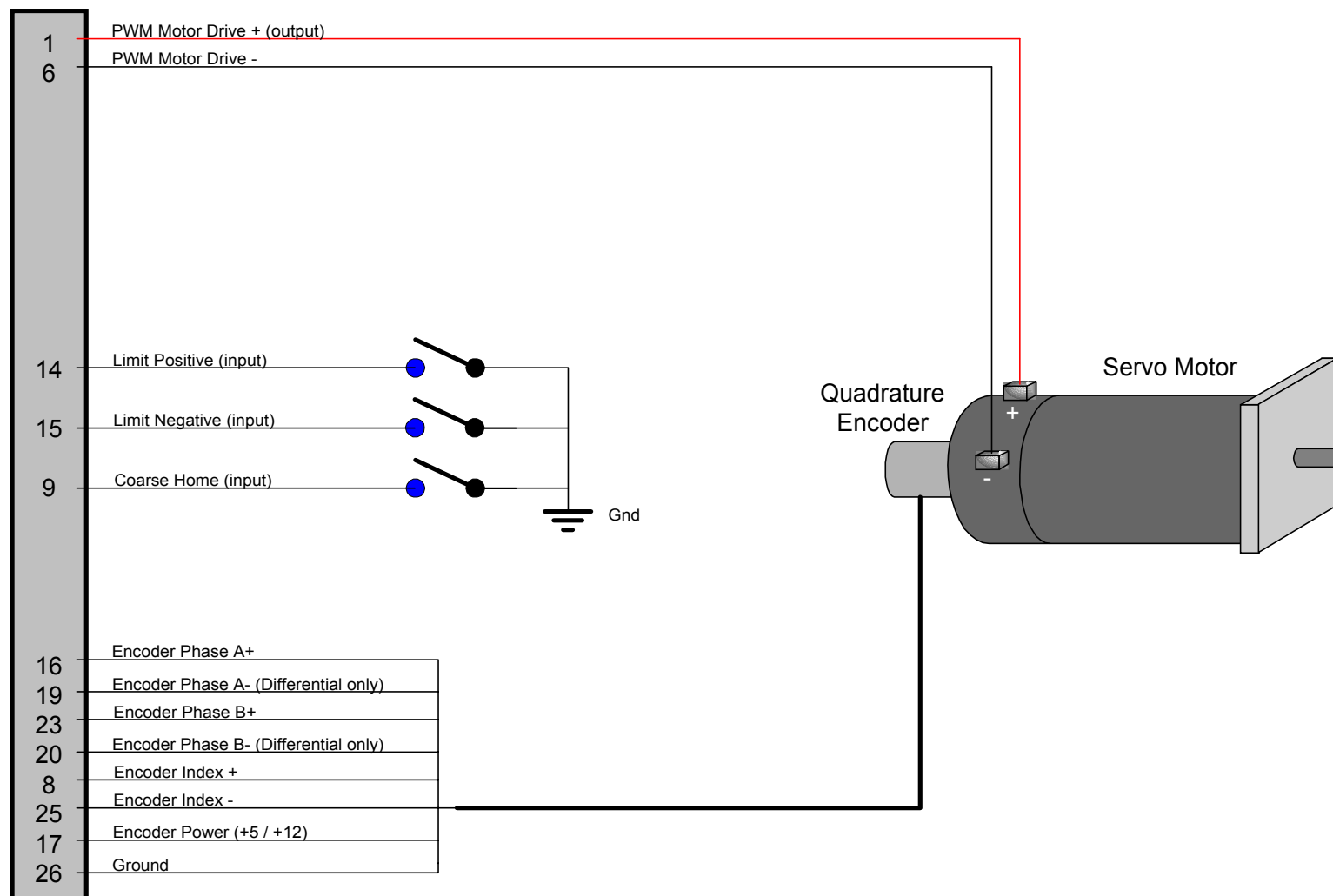
Jumper	Setting	Encoder Power Supply (JP3 pin 17)
JP3	Pins 1 to 2	+5VDC
JP3	Pins 2 to 3	+12 VDC
JP3	Open	Open



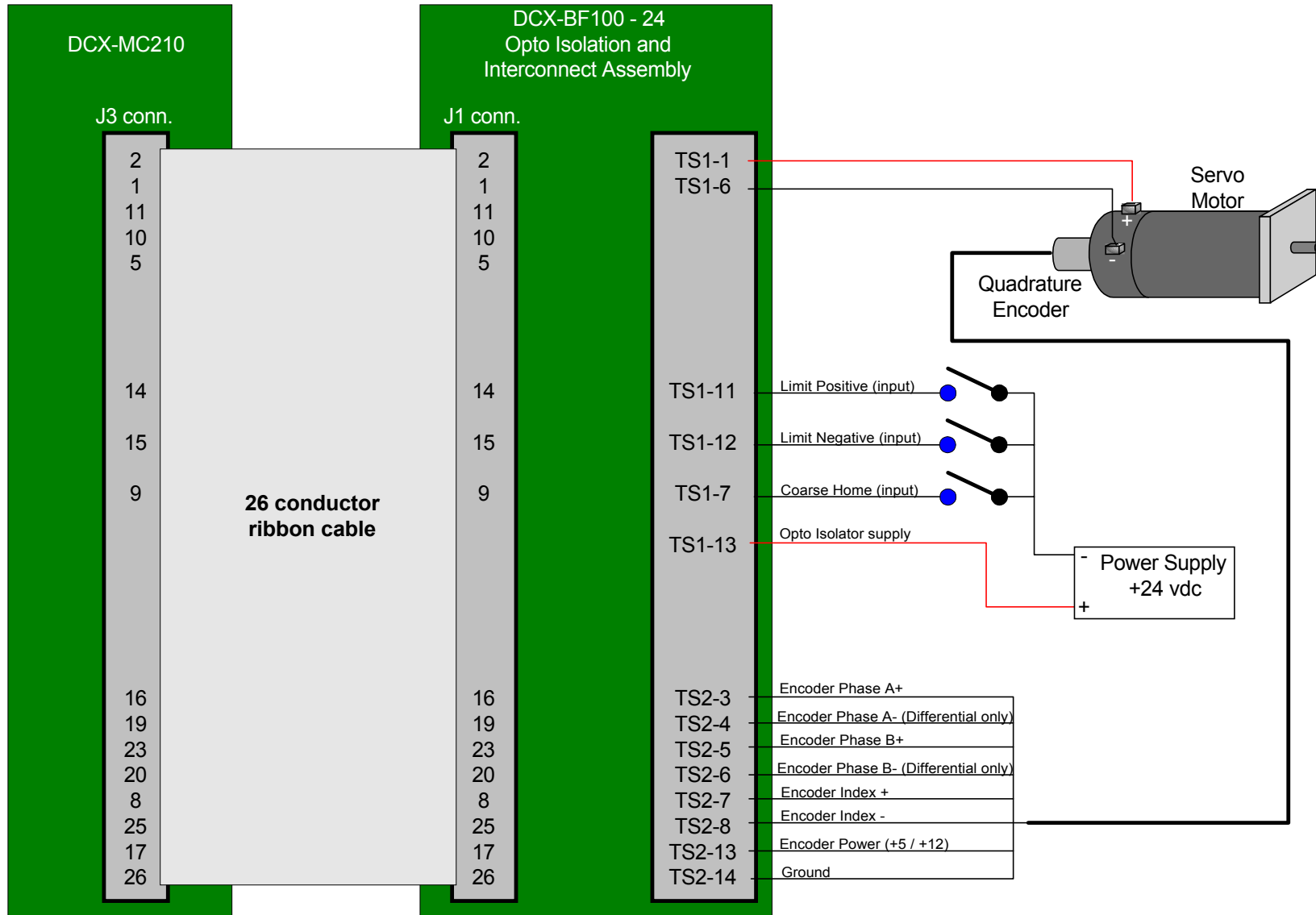
**Note:** The DCX-MC210 provides the Encoder Power output as a convenience, It is **not required** that it be used to power the encoder. If an external +5 volts or +12 volts supply is used to power the encoder, jumper JP3 **must still be configured** to match the voltage level (+5 volts or +12 volts) of the external supply.

After configuring the jumpers of the module, the servo 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 MC210 to the external components (servo motor, encoder, and sensors). The second diagram details typical connections when the **DCX-BF100 Opto Isolation and Interconnect Assembly** is used.

DCX-MC210  
I/O Connector J3



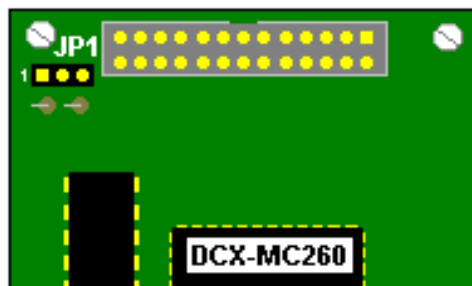




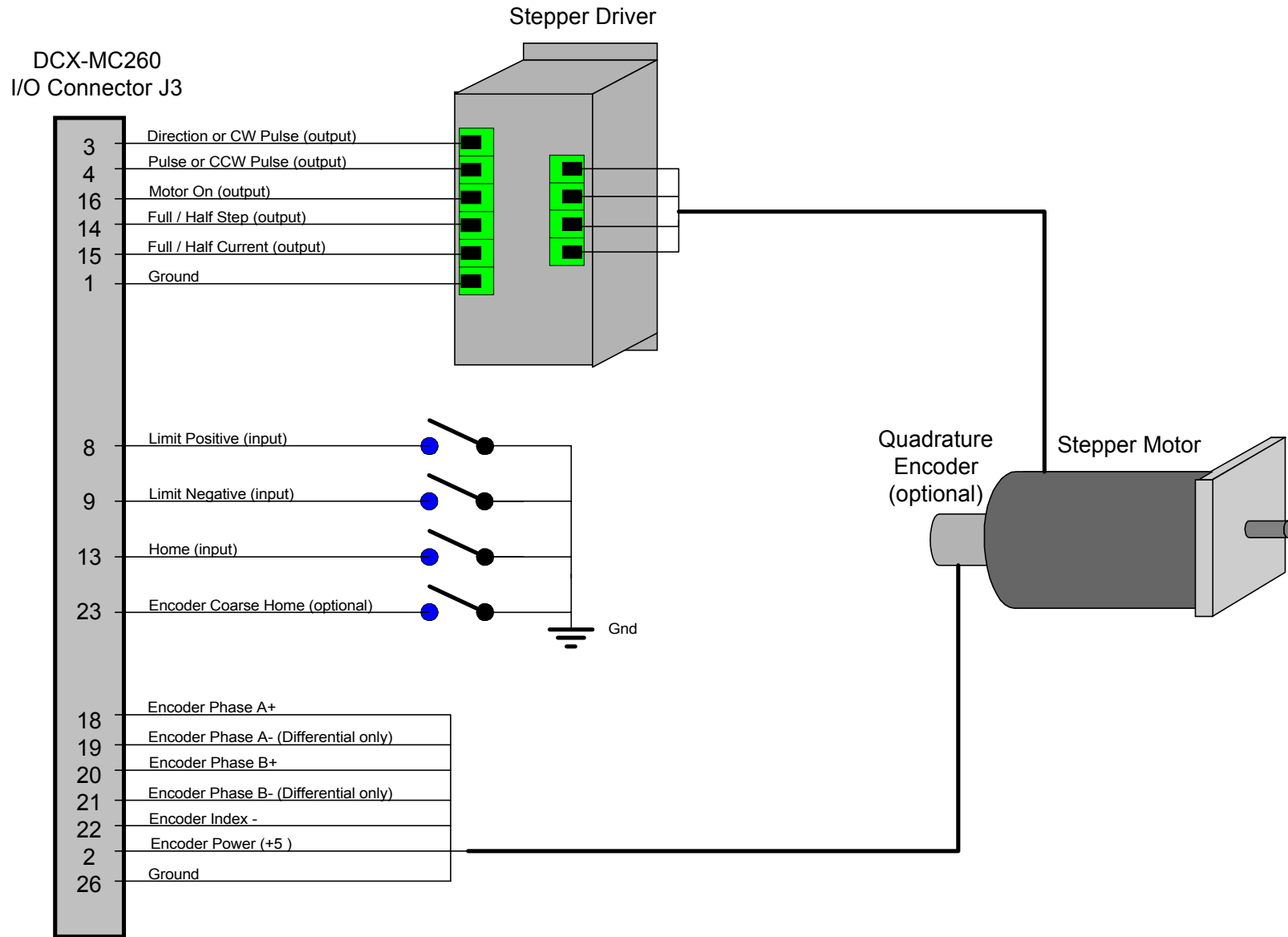
## DCX-MC260 – Stepper Motor Module Installation

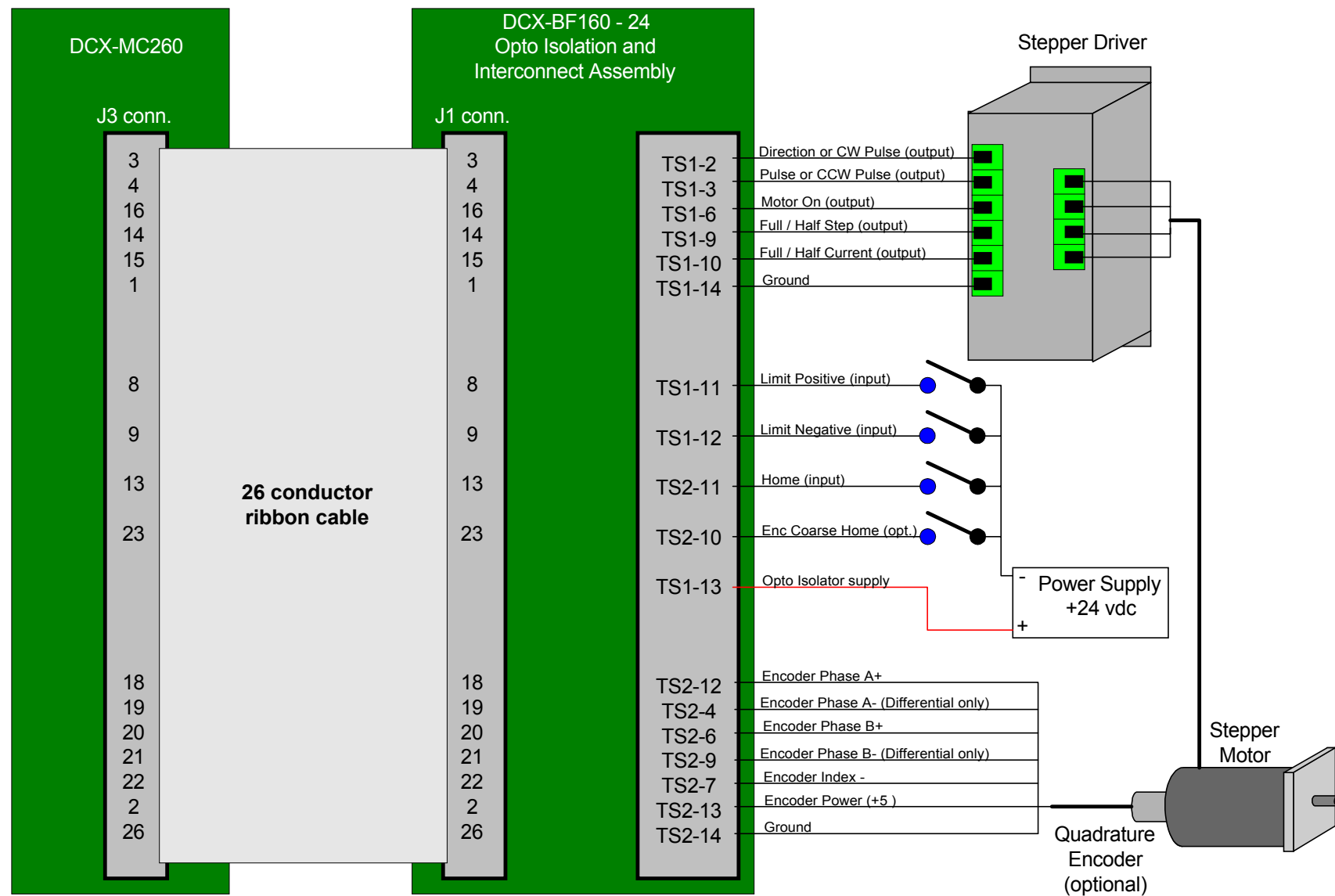
Installation of a DCX-MC260 Stepper Motor Control Module includes setting jumper JP1 if an incremental encoder will be used for position feedback.

Jumper JP1 configures the module's encoder phase inputs for 'single ended' (A and B) or 'differential' (A+, A-, B+, and B-) signals. For single ended outputs, a 3 hole shorting block (supplied with the module) should be installed across all 3 pins of jumper JP1. In this case, the A and B signals from the encoder should be connected to the A+ and B+ inputs of the module. If an encoder with differential phase outputs is to be connected to the module, jumper JP1 should be left open (no shorting block installed).



After configuring the jumper of the module, the stepper driver, limit switches and optional encoder can be connected to the module. Wiring diagrams on the next two pages depict typical installations. The first diagram details direct connection of the MC260 to the external components (stepper driver, sensors, and optional encoder). The second diagram details typical connections when the **DCX-BF160 Opto Isolation and Interconnect Assembly** is used.

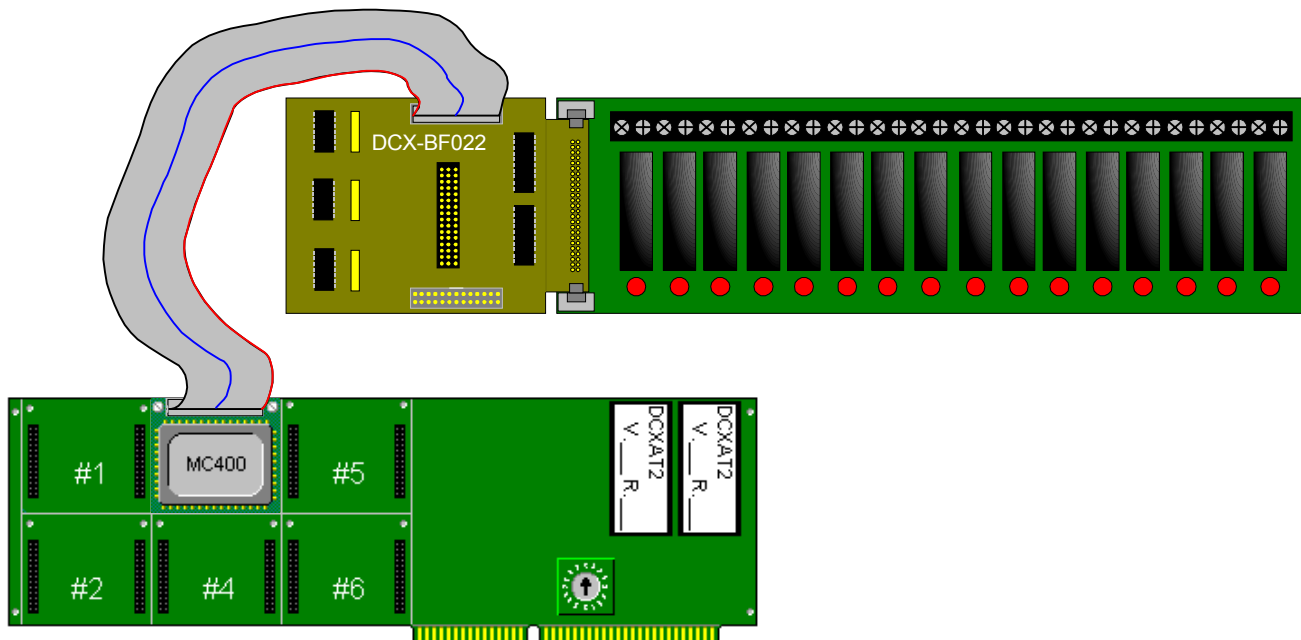




## DCX-MC400 – Digital I/O Expansion Module Installation

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.

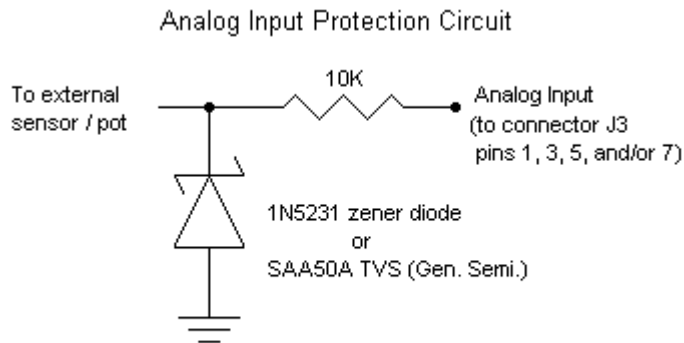
For detailed information on configuring the DCX-BF022, please refer to the schematic and jumper table in the DCX-BF022 Appendix in this user manual.

## 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.



## DCX-MF300 – RS-232 Stand Alone Communication Module Installation

A single MF300 RS-232 module can be installed in any module position on the DCX. There are several jumpers on the module that should be configured before connecting the module to an external device. These jumpers are configured by installing shorting blocks on the pins of the jumpers, or leaving them open. An appendix of this manual which covers the RS-232 module, includes a description of these jumpers and a diagram showing their locations. In addition, a portion of the module schematic is included in the appendix to help in configuring the module. Note that the pins of each jumper are numbered sequentially from 1, with pin 1 being shown as a square.

The RS-232 module connector J3 pin-out, is designed to allow a cable to be assembled from ribbon cable and Insulation Displacement Connectors (IDC). Such a cable, with a 25 pin D-subminiature connector on one end, can be directly connected to a terminal or host computer serial port. If the external device is considered to be Data Terminal Equipment (DTE), then the RS-232 module should

be configured as Data Communications Equipment (DCE). This is the default configuration of the RS-232 module as shipped from the factory, and should have the jumpers set as follows:

<b><i>Jumper number</i></b>	<b><i>Default setting</i></b>
JP1	Pins 9 to 10
JP2	Pins 1 to 3, 2 to 4
JP3	Pins 1 to 2
JP4	Pins 1 to 2
JP5	Open
JP6	Pins 1 to 2
JP7	Open
JP8	Pins 1 to 2
JP9	Open
JP10	Pins 1 to 2
JP11	Open
JP12	Open
JP13	Open
JP14	Open

Jumper JP2 on the RS-232 module is used to select whether internal control signals are used for hardware handshaking, or for networking. When a single DCX is connected to the communicating device, this jumper should be configured for handshaking. This is done by installing a shorting block on pins 1 and 3, and another on pins 2 and 4 (shorting blocks will be perpendicular to 'JP2' label). For configuration of the module for use on a RS-232 network, a shorting block should be installed on pins 1 and 2, and another on pins 3 and 4.

## DCX-MF310 – IEEE-488 Stand Alone Communication Module Installation

A single MF-310 IEEE-488 module can be installed in any module position on the DCX. There are two jumpers on the module that should be configured before connecting the module to an external device. These jumpers are configured by installing a shorting block on the pins of the jumpers, or leaving them open. An appendix of this manual which covers the IEEE-488 module includes a description of these jumpers and a diagram showing their locations.

The IEEE-488 module connector J3 pin-out, is designed to allow a cable to be assembled from ribbon cable and Insulation Displacement Connectors (IDC). Such a cable, with an IEEE-488 connector on one end, can be directly connected to a IEEE-488 controller.

Jumper JP1 should have a shorting block installed to provide open collector drivers on the module. Leaving the pins of jumper JP1 open will configure the module drivers as push-pull for high speed communications.

Installing a shorting block on jumper JP2, will connect the IEEE-488 connector's shield signal to the DCX's ground. Since the IEEE-488 controller should provide the grounding point for the cable shields, this jumper should be left open. The other jumpers on the IEEE-488 module are for production options, and should be left as shipped from the factory (JP3 hardwired, JP4 and JP5 open).

The six position DIP switch on IEEE-488 module is used to set the address of the DCX on the IEEE-488 bus. In the **DCX-MF310** section of the **Connectors, Jumpers, and Schematics** chapter a table lists all possible address settings. As an example, to set the DCX's address to 4 (talk address = 'D', listen address = '\$'), switches 1, 2, 4, 5 and 6 should be in the OFF position, and switch 3 should be in the ON position.





## Chapter Contents

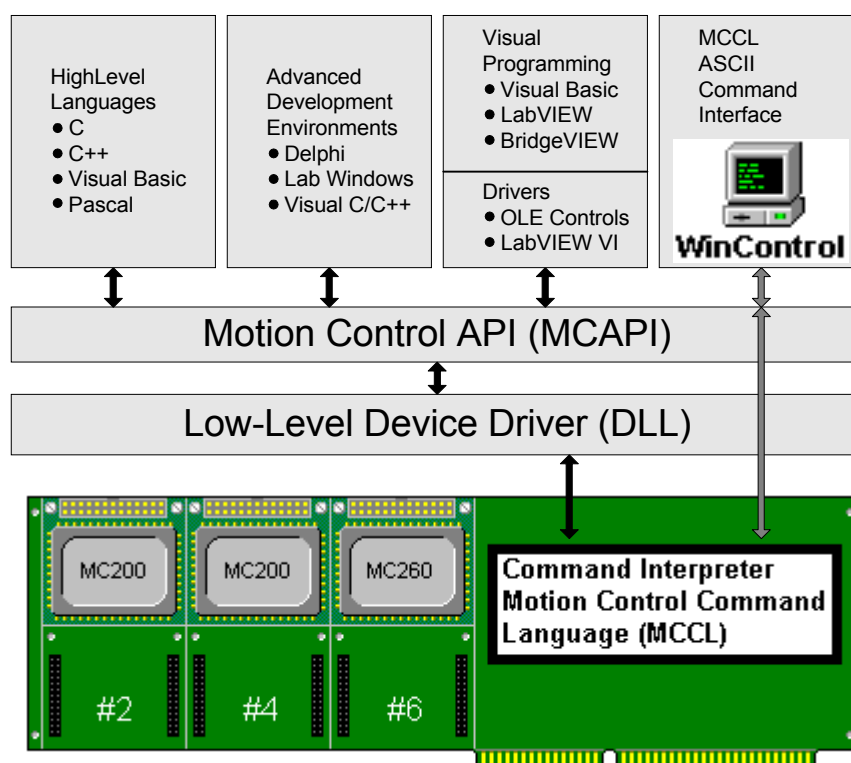
---

- 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

## Programming, Software and Utilities

The DCX motion control system is engineered to integrate seamlessly into high performance, PC based, Windows applications. Support for all popular high level languages is provided by the Motion Control API. Additionally, a powerful set of board level commands are available, allowing the motion controller to execute local 'macro' routines independent of the PC host and application program.

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.



The API has been constructed with a layered approach. As new versions of the Windows operating system and new motion controllers become available it will be possible to provide API support by 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 systems.

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! See the Programming section for additional information on adapting the API to other development environments.

## 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. The CWDEMO and VBDEMO sample programs 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 open handle 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.

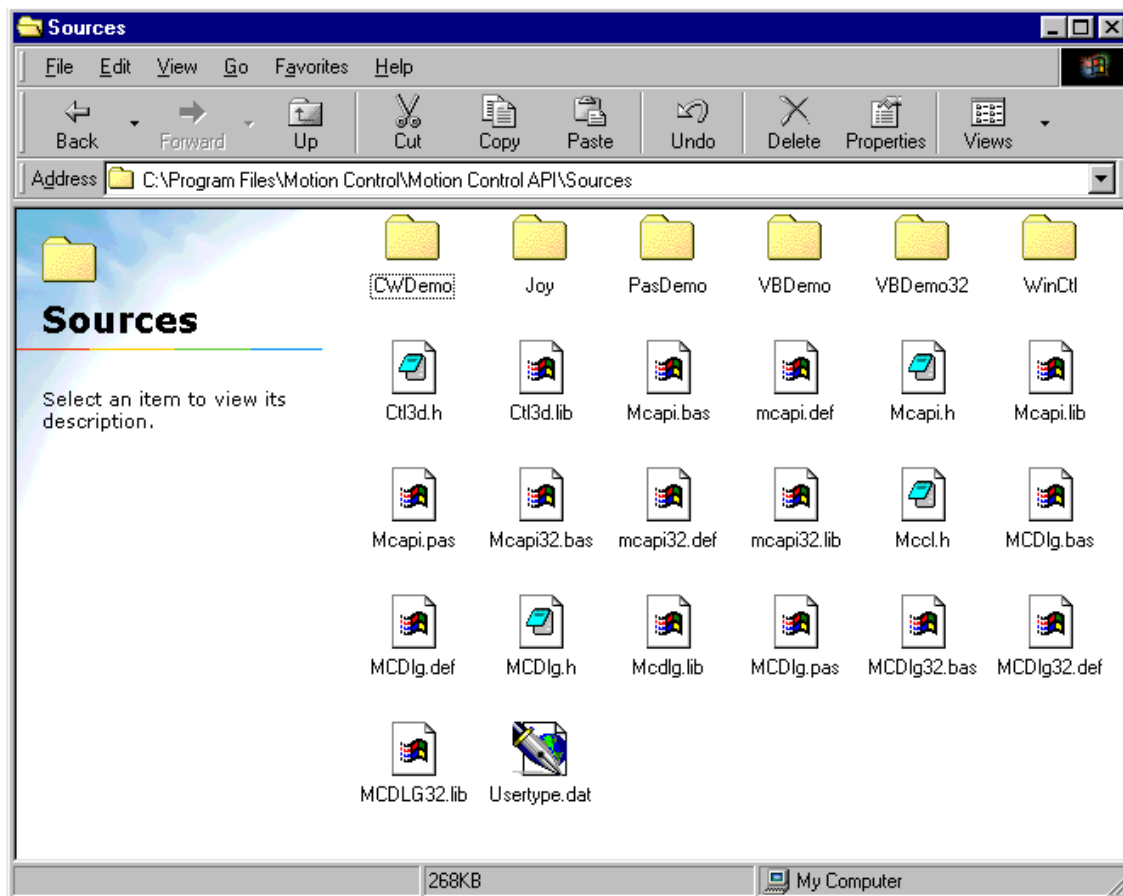
## Building Application Programs using Motion Control API

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/step units, velocity units, dwell time units, user and part zero)
- Commanding motion (Point to Point, Constant velocity, Electronic Gearing, Lines and Arcs, Joystick control)
- 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 chapter 9, **Motion Control Function Reference**.

Included with the installation of the Motion Control API is the Sources 'folder'. In this folder are complete program sample source files for C++, VisualBasic, Delphi.



## 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

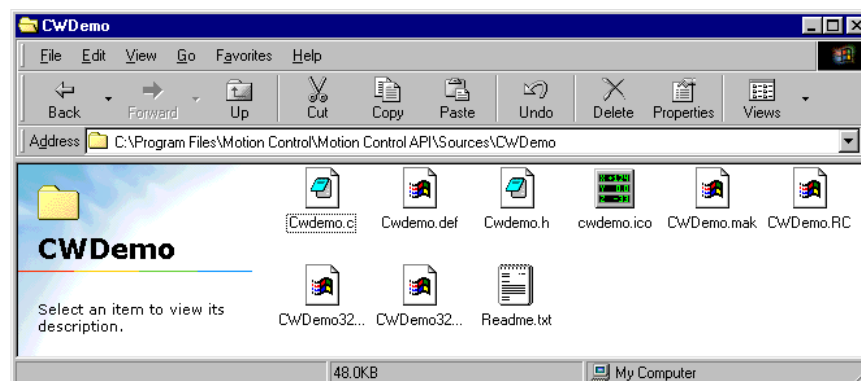
If your compiler does not define the `_WIN32` constant for 32-bit projects you will need to define it at the top of the source file (before the header files are included).

### 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

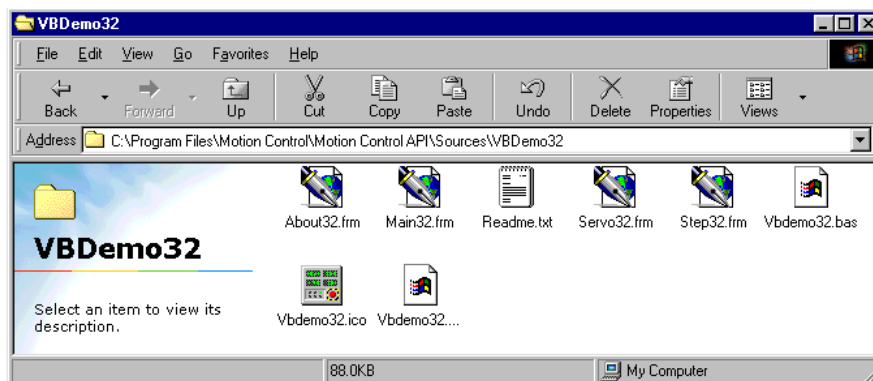
=====

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 (Pdmo.dpr for 16-bit, Pdmo32.dpr for 32-bit). Include the following files if you create your own project:

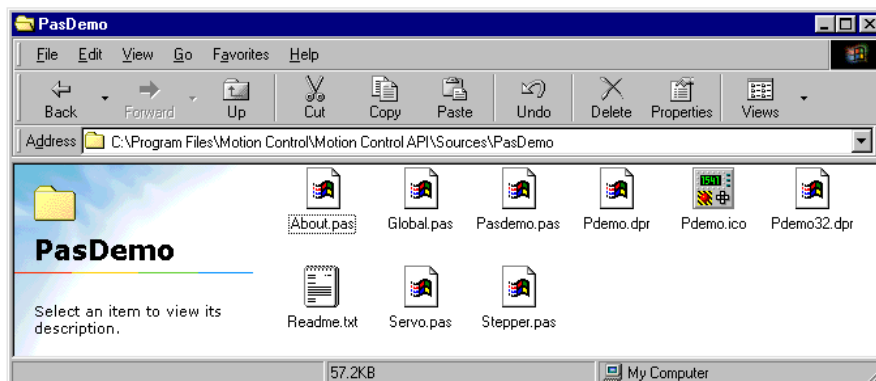
About.pas  
Global.pas  
PasDemo.pas  
Servo.pas  
Stepper.pas

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

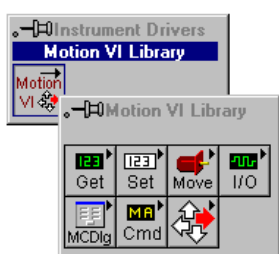
PMC's LabVIEW Virtual Instrument Library includes an On-Line help with a Getting Started guide.

**Motion VI Library Help**

File Edit Bookmark Options Help

Contents Index Back Print << >>

### Getting Started



Before you install the Motion VI Library you must first install LabVIEW version 5.0 for Windows 95 / 98 / NT. This is necessary so that the Motion VI Library can add its function and control palettes to the LabVIEW menu system, and install the online help where LabVIEW can locate it.

You also need to have the 32-bit Motion Control API (MCAPI) installed and configured before you can begin using the Motion VIs. The current MCAPI release is available from the PMC World Wide Web site and may be installed before or after you install the Motion VI Library. For full functionality you must use MCAPI version 2.1c or higher.

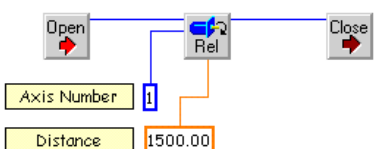
#### Samples

Four sample programs are now included with the Motion VI library. The first, **SIMPLE.VI**, shows how to execute a simple move. The **SAMPLE.VI** sample provides an interactive panel for moving an axis and monitoring the status of that axis. **CYCLE.VI** demonstrates how to implement a state machine and execute multiple moves under program control (the state machine approach makes it easy to monitor the status of axes while the motions are executed). Finally, **ANALOG.VI** demonstrates the use of the auxiliary analog inputs available on most PMC motion controllers.

The Motion VIs are installed in the Instrument Drivers function palette in a number of logically arranged sub-palettes. To better see how the VIs are used, open the **SAMPLE.VI** from the file menu (select File | Open, select the INSTR.LIB directory, then the MOTION CONTROL directory, and finally **SAMPLE.VI**).

The first step in any motion program is to obtain a handle to the controller, using the **MCOpen** VI. This handle is used in all subsequent calls to the Motion VIs. When the program completes the handle should be passed to the **MCClose** VI to ensure the motion controller is properly closed. Failure to properly close the handle is the primary source of errors when using the Motion VI Library. The following wiring diagram, from the **SIMPLE.VI** sample program, demonstrates how to open the motion controller, perform a simple move, and close the motion controller:

Minimal motion sample - opens a motion controller, moves axis one 1500.0 counts in the positive direction, and closes the handle.



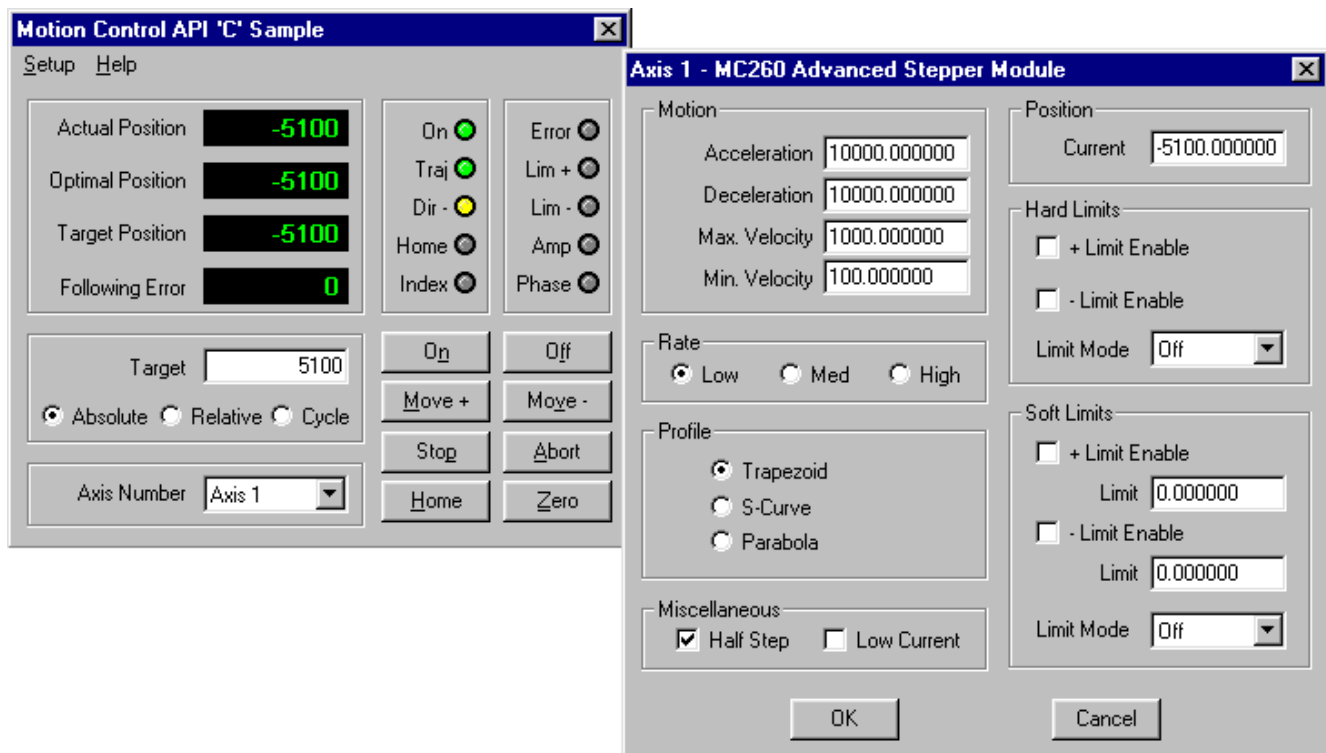
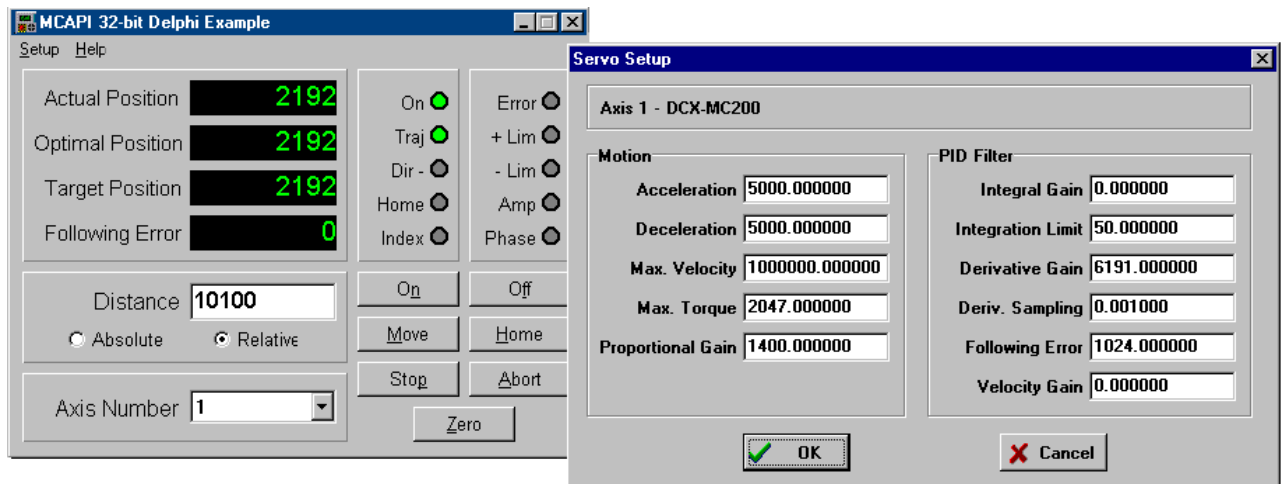
DCX-AT200 User's Manual

47

## 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 (servo or stepper)
- 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

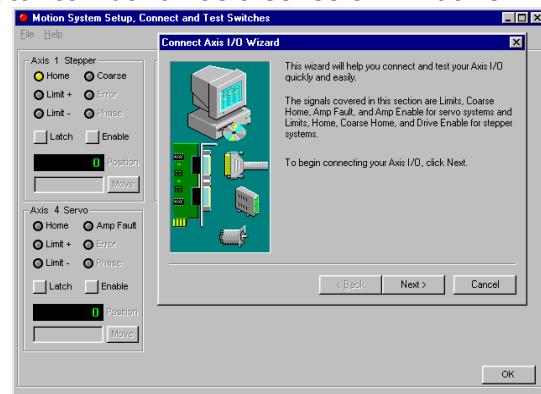
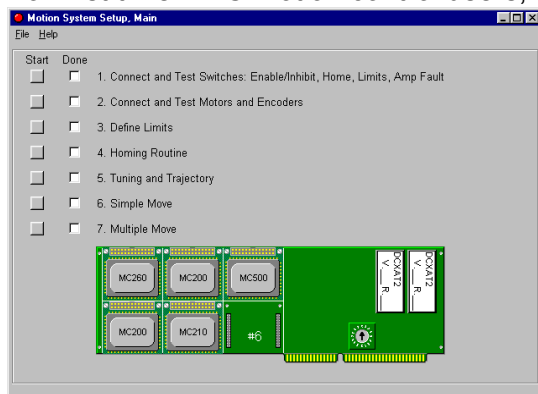


# Motion Integrator

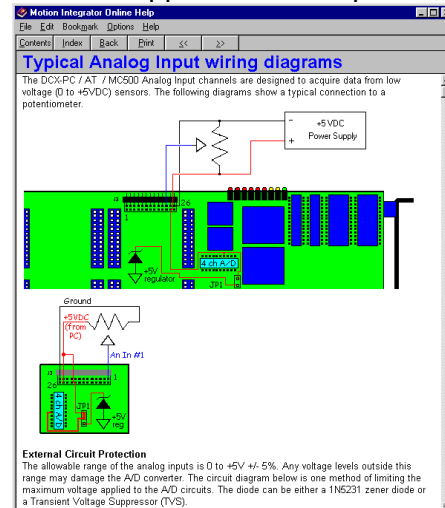
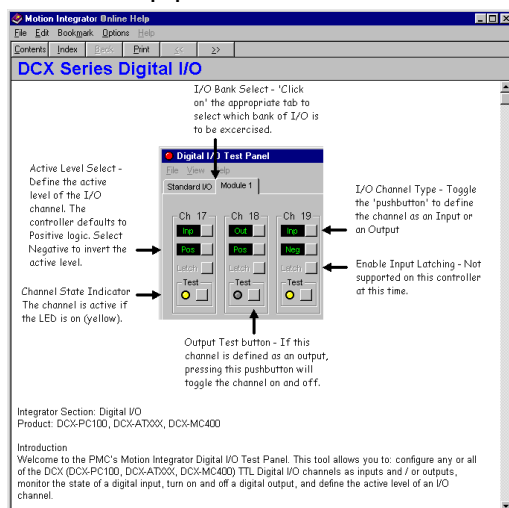
PMC's new Motion Integrator program is just like having your own 'Systems Integrator' to assist you with every step of the integration process. Motion Integrator is a suite of powerful Windows tools that are used to:

- 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



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



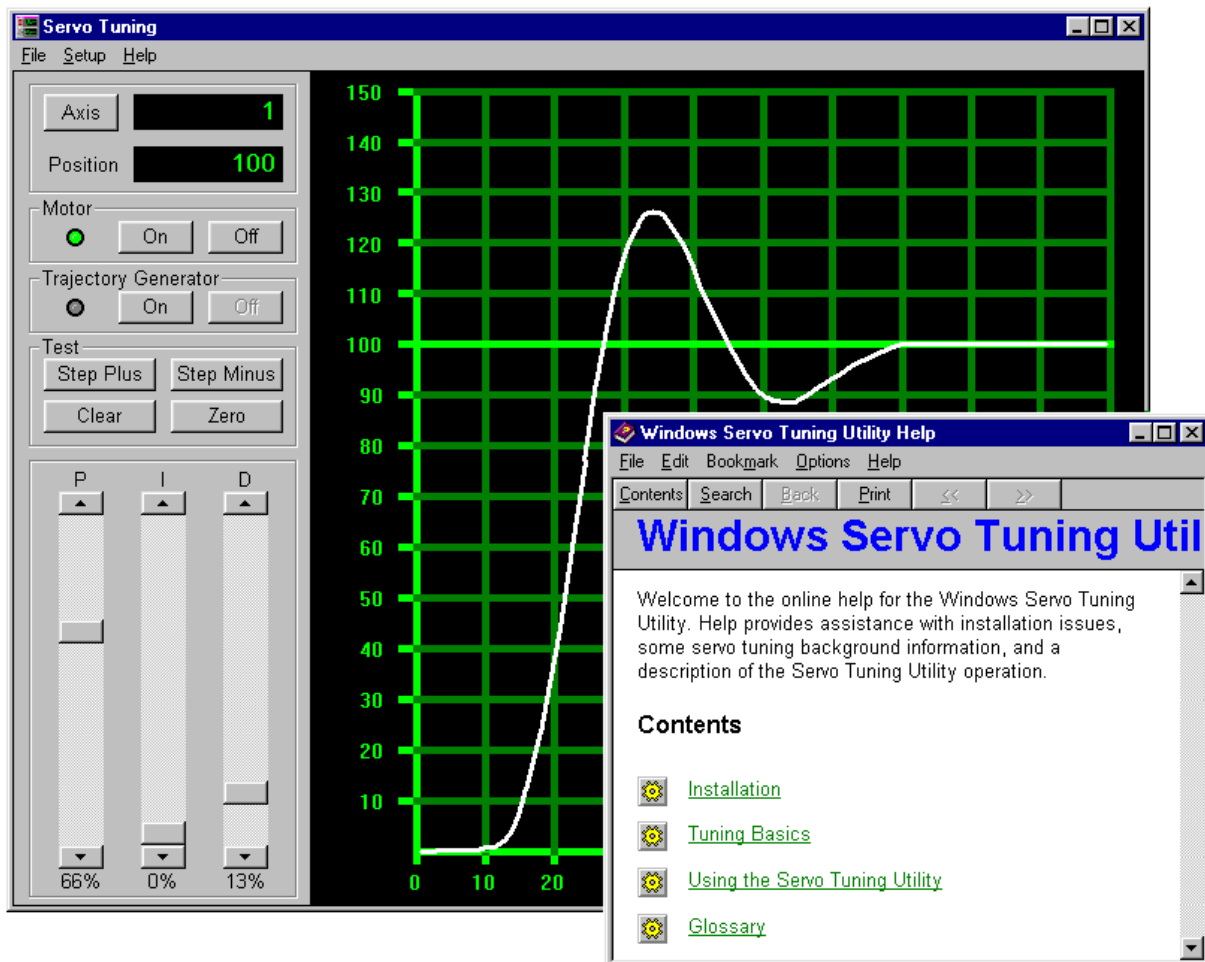
### Tuning servo's with Motion Integrator

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

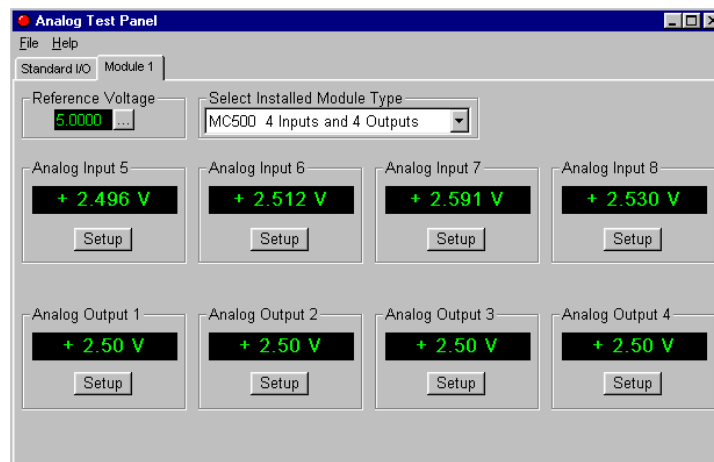
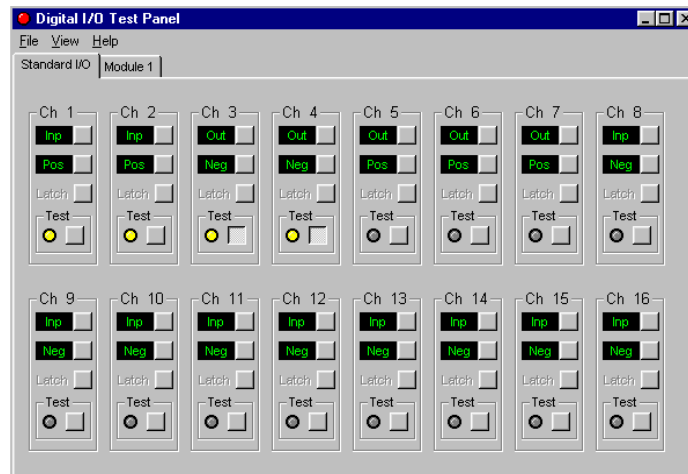
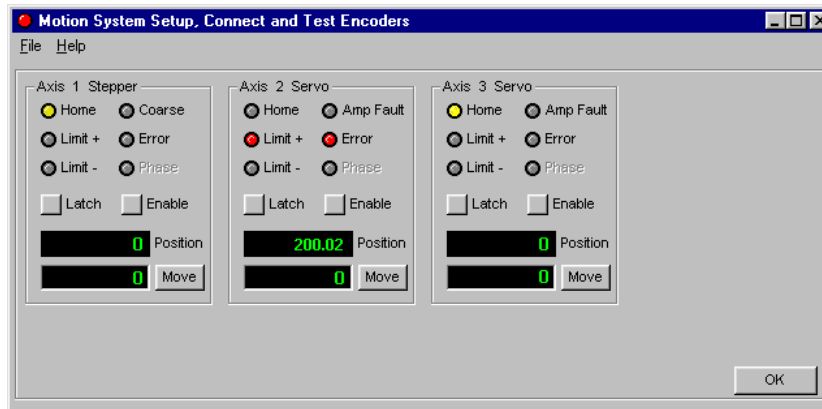
With the Trajectory generator turned on, the user can execute 'real world' moves displaying the calculated position, actual position, following error, and DAC output plots.



The Servo Tuning Utility on-line help provides assistance both in using the utility and it provides a 'primer' describing basics of servo tuning.

## Motion and I/O Test Panels are Embeddable OLE Servers

The Motion Integrator test panels (Axis I/O, Digital I/O, and Analog I/O) allow the user to easily exercise the system outside the application program. These OLE Server tools can be embedded into the user's application program for system diagnostics.



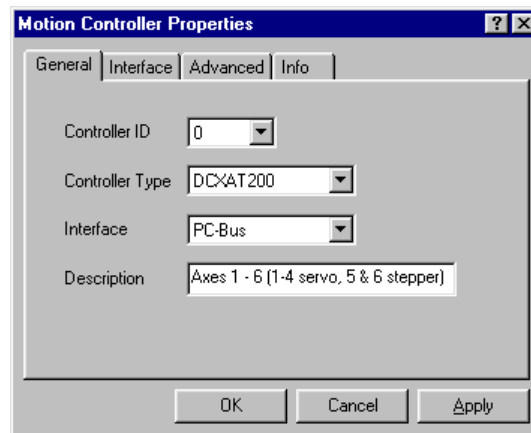
## PMC Utilities

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

- Configure the Motion Control API for the type and quantity of DCX controllers
- 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

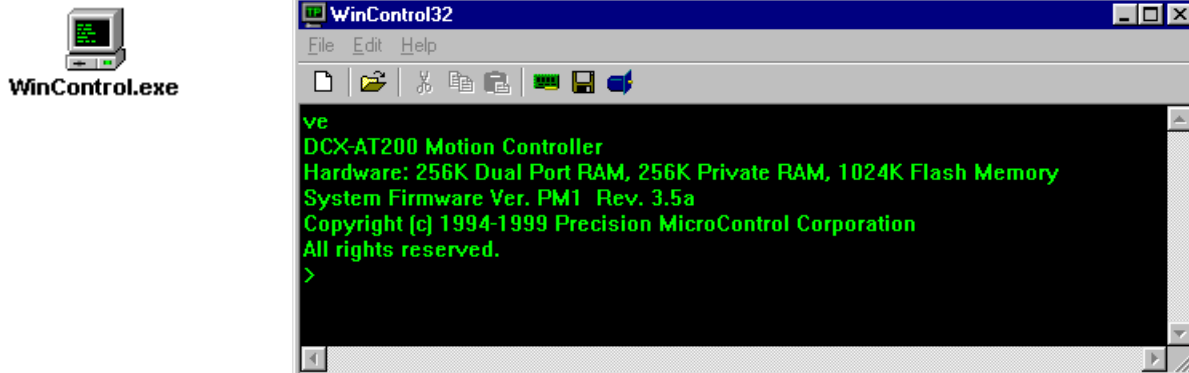
### The MCAPI Setup Utility

This utility is used to configure the MCAPI for the type, address, and quantity of DCX control cards. 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 in the native language (MCCL) of the controller. 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 DCX 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



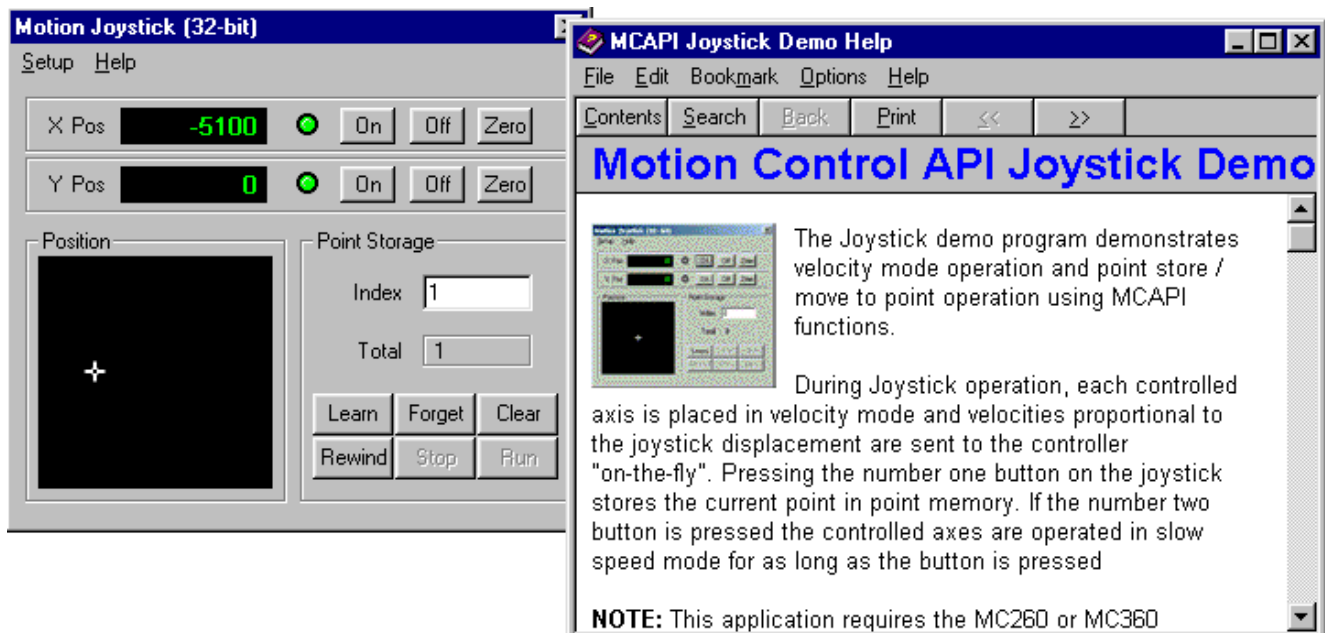
## Flash Wizard

PMC's Flash Wizard provides the user with an easy to use tool for 'field' upgrades of the Flash based DCX firmware.



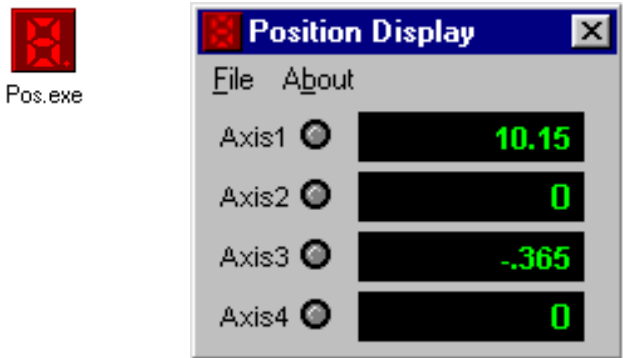
## Joystick Applet

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.



**Position Readout**

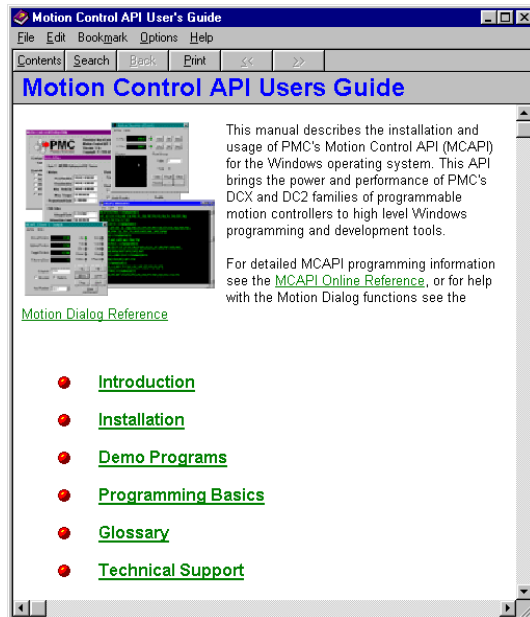
This MCAPL utility will report the position of all installed motor control modules.



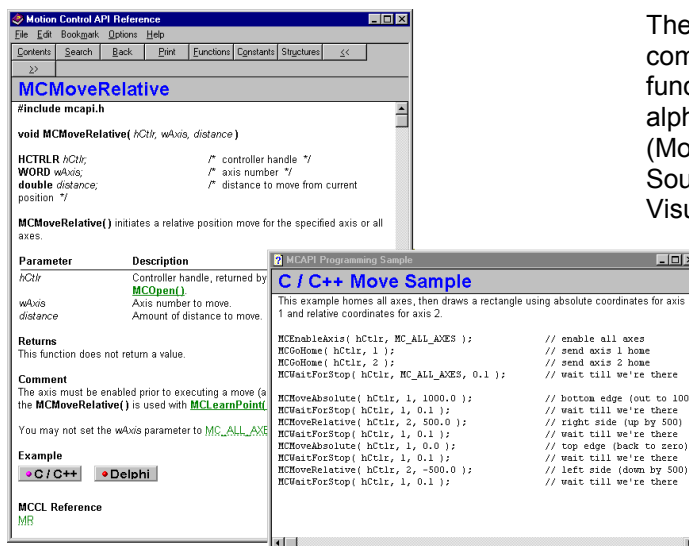


# MCAPI On-line Help

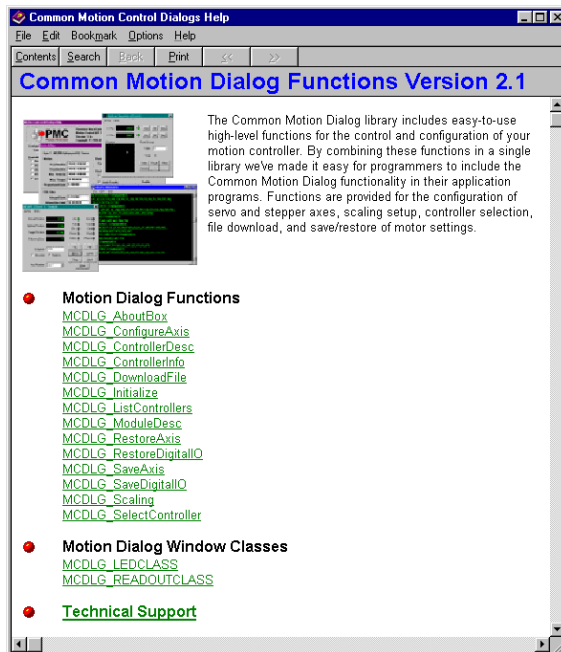
Complete and up to date (from PMC website [www.pmccorp.com](http://www.pmccorp.com)) On-line help for PMC's MCAPI (Motion Control Application Programming Interface). Help documents include; installation and basic usage, complete function call reference and examples, high level dialog descriptions, and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



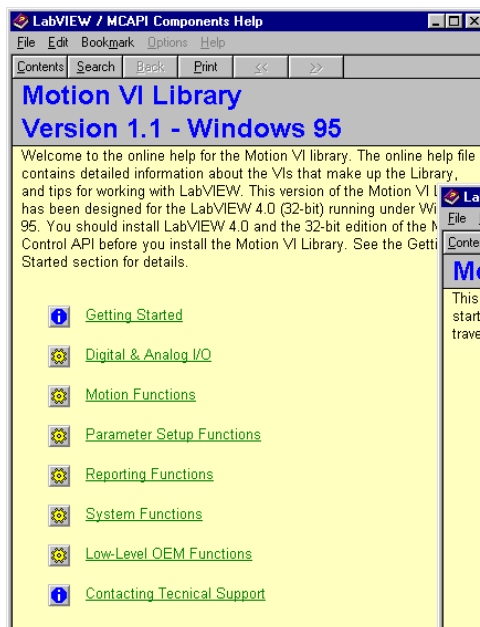
The MCAPI Users Guide On-line Help describes the basics of PMC's MCAPI. This should be the **'first stop'** for any questions about the MCAPI.



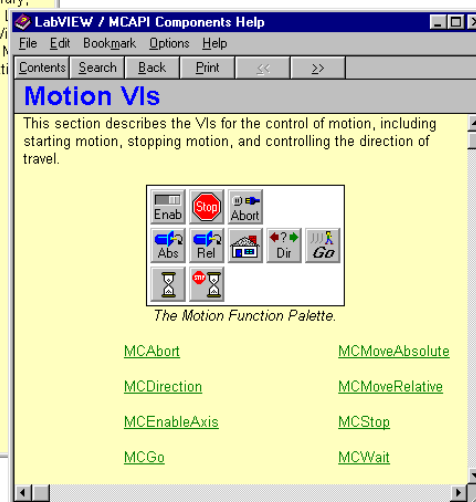
The MCAPI On-line Help provides a complete listing and description of all MCAPI functions. Function calls are grouped both alphabetically and by functional groups (Motion, Setup, Reporting, Gearing, etc...). Source code examples are provided for C++, Visual Basic, and Delphi.



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.



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





## **Chapter Contents**

---

- PC Communication Interface
- RS-232 Communications Interface
- IEEE-488 Communications Interface

## Communication Interfaces

---

The DCX controller provides a high speed binary interface for communicating with the PC via the ISA bus. This interface is implemented using dual ported memory that is mapped into the PC using a single address switch.

An ISA ASCII communication interface is also provided. This communication port allows the user to communicate directly with the DCX in its native language, MCCL (Motion Control Command Language).

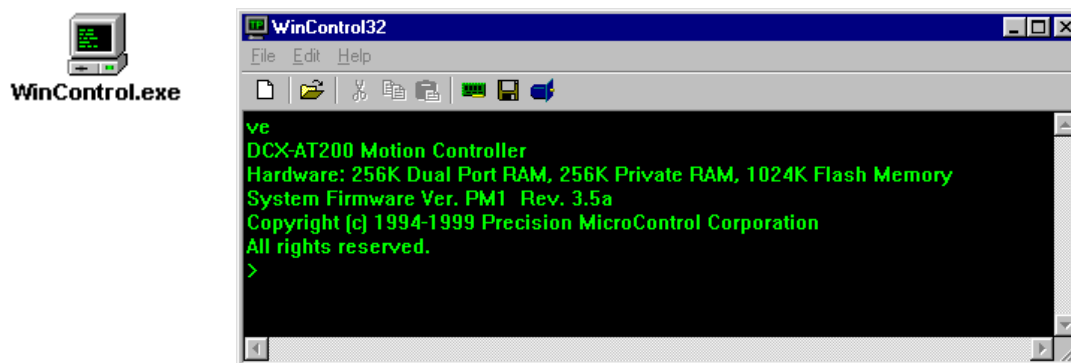
The DCX board also supports optional auxiliary communications interfaces. The optional RS-232 serial interface is supported by installing the DCX-MF300 RS-232 interface module available from PMC. For interfacing to the DCX board over the IEEE-488 Bus, the MF310 IEEE-488 module is available from PMC.



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.

## PC communications Interfaces

With the DCX board installed in an Intel compatible PC, the WinControl Terminal Emulator implements MCCL (Motion Control Command Language) low level communications with the DCX through the host ASCII interface. This 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](http://www.pmccorp.com)



In addition to sending command lines from the keyboard and displaying responses on the host display, this program can be used to send a text file containing MCCL commands 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.

Most PC based motion control applications require high speed communications between the host and the controller. PMC's MCAPI provides C++, Visual Basic, Delphi, and LabVIEW drivers for PC applications programmed in high level languages. For additional information about available software and integration tools please refer to the **Programming, Software, and Utilities** chapter of this manual.

The MCAPI drivers are implemented using the 'DCX Binary Command Interface'. This interface uses binary formatted commands and replies that provide the most efficient means for the host to communicate with the DCX. The DCX Binary Command Interface is described in full detail in the **Appendix** at the end of this manual. In some situations, it is preferable for the host to send commands to the DCX in an ASCII character format, and for the board to send replies as ASCII characters. This is accomplished using the 'ASCII Command Interface' of the DCX. This interface is described in full detail in the **Appendix** found at the end of this manual.

## RS-232 Communications Interface

The RS-232 interface has been designed to use either software or hardware handshaking for its operation. Hardware handshaking can be enabled or disabled by use of the HN and HF commands. Software handshaking can be enabled or disabled by use of the XN and XF commands.

Set the configuration of the RS-232 host to 8 data bits, one stop bit and no parity. This is the factory default configuration of the RS-232 interface. Also set the device to translate carriage returns received

into carriage return and linefeed. The baud rate of the host should be set to match the baud rate of the RS-232 module (9600 baud is the default).

Note that input characters are only echoed through the RS-232 interface when enabled by the Echo ON (EN) command.

## **IEEE-488 Communications Interface**

In order to send commands to the DCX via the IEEE-488 Bus interface it must first be addressed to listen. The command line is then sent in ASCII format, including the ending carriage return (no line feed). The DCX should then be unaddressed as a listener and addressed as a talker. If the DCX has a response to the command ready, it will send it at that time. Note that the DCX will not accept a new command line until the responses to the previous command have been accepted.

## **Chapter Contents**

---

- Introduction
- Commanding DCX Operations



## 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 which it can perform. This instruction set, known as MCCL (Motion Control Command Language), consists of over 200 operations which 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:

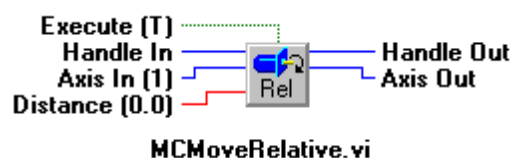
### Move to relative position

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:** MC200, MC210, MC260  
**see also:** Move to absolute position

**C++ Function:** void MCMoveRelative( HCTRLR hCtrlr, WORD wAxis, double Distance );  
**Delphi Function:** procedure MCMoveRelative( hCtrlr: 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***.

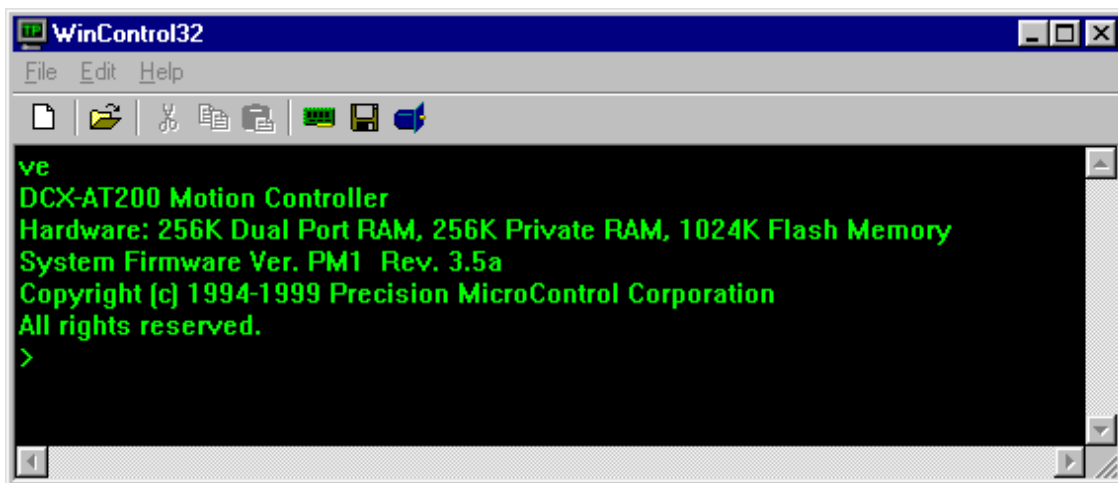
## Low Level DCX Operations

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 via any of the supported interfaces (ISA host ASCII, RS-232, and IEEE-488). Each MCCL command is described in detail in the **DCX MCCL Command** section of the **Appendix** of this user manual.

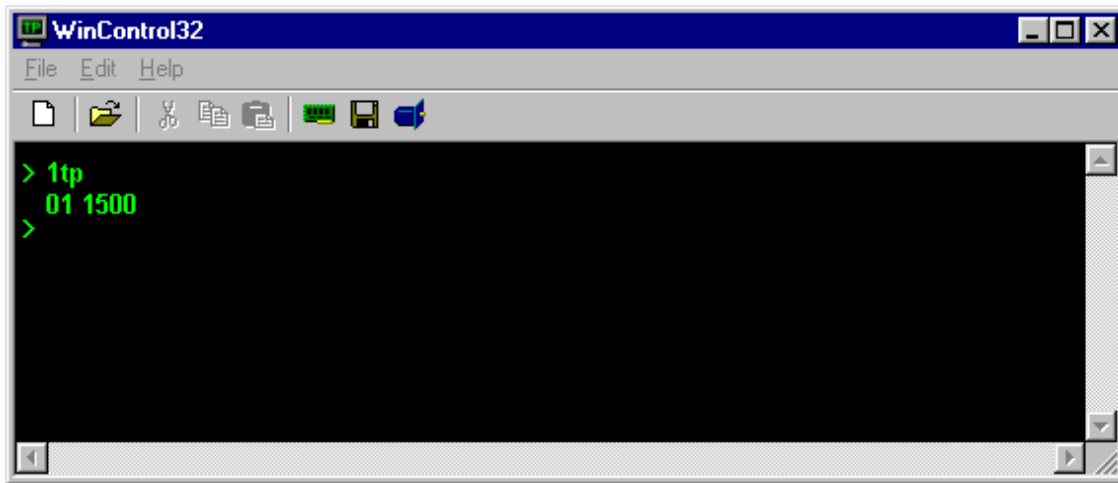


**Note** – Prior to issuing any commands to the DCX via WinControl, the **MCAPI Setup utility must be configured for type, address, and quantity** of DCX controllers. See the description of the MCAPI Setup utility in the **Installation and Programming, Software, and Utilities** chapters.

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 (this requires Echo oN for the RS-232 interface). 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 **Appendix** 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, and deceleration), allowable following error (servo only), 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, Servo Operation and Stepper 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 **Tell Target** (aTT) and **Tell Position** (aTP) commands should report the same number. There are several commands which are used to begin motion, including **Move Absolute** (MA) and **Move Relative** (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 **RePeat** (RP) command at the end causes the previous command to be repeated 6 times. The **Wait for Stop** (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)
```

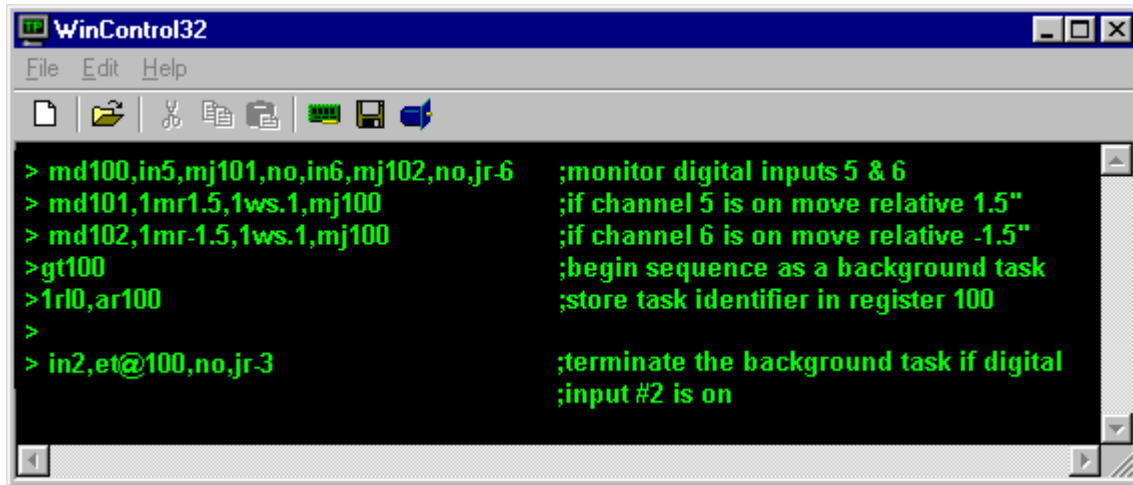
The DCX will respond with a succession of numbers indicating the position of the axis at that time. Many terminals have an "auto-repeat" feature which allows you to track the position of the axis by simply holding down the return key.

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.



The screenshot shows a window titled "WinControl32" with a menu bar (File, Edit, Help) and a toolbar. The main area is a black text field with green text. The text is organized into two columns: commands on the left and their descriptions on the right. The commands are preceded by a green prompt character '>'. The descriptions are preceded by a semicolon ';'. The commands and descriptions are as follows:

Command	Description
> md100,in5,mj101,no,in6,mj102,no,jr-6	;monitor digital inputs 5 & 6
> md101,1mr1.5,1ws.1,mj100	;if channel 5 is on move relative 1.5"
> md102,1mr-1.5,1ws.1,mj100	;if channel 6 is on move relative -1.5"
> gt100	;begin sequence as a background task
> 1rl0,ar100	;store task identifier in register 100
>	
> in2,et@100,no,jr-3	;terminate the background task if digital input #2 is on

## **Chapter Contents**

---

- Theory of DCX Motion Control
- DCX Servo Basics
- Tuning the Servo
- DCX Stepper Basics
- Closed Loop Steppers
- Moving Motors with PMC demo's
- Defining the Characteristics of a Move
- Velocity Profiles
- Point to Point Motion
- Constant Velocity Motion
- Contour Motion (arcs and lines)
- Electronic Gearing
- Jogging
- Defining Motion Limits
- Homing Axes
- Motion Complete Indicators
- On the Fly Changes
- Feed Forward (Velocity, Acceleration, Deceleration)
- Save and Restore Axis Configuration

## Motion Control

---

This chapter describes the basic building blocks of DCX motion control. In general, the modes of motion described in this chapter are common to both servo and stepper motors, with specific differences detailed in the text.

### Theory of DCX Motion Control

The DCX motherboard (DCX-AT200) uses a 32 bit RISC processor which is programmed to perform motion control tasks. Specially designed servo or stepper motor control modules are installed on the motherboard to configure it for controlling from 1 to 6 servos or stepper motors. Each DCX motion control module (DCX-MC200, DCX-MC210, DCX-MC260) 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...).

The motherboard processor implements a trajectory generator (trapezoidal, S curve, and parabolic) which calculates the desired position and velocity of each servo or stepper motor at fixed time intervals. These values are sent to the respective servo (DCX-MC200 or DCX-MC210) or stepper module (DCX-MC260) installed on the motherboard. Each servo or stepper module has a 16 bit processor which is programmed to provide the appropriate control of the servo or stepper motor interfaced to the module.

#### Servo Motor Control

The DCX servo modules use a velocity feed-forward and a position feedback loop to control the servo. The **DCX-MC200** uses a 12 bit, +/-10 volt analog output signal to an external servo amplifier. The **DCX-MC210** provides a 23.4 KHz, 8 bit, PWM direct motor drive output capable of driving a 12 volt motor with up to 1A 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 and the velocity feed-forward are combined to set the

module's analog output level. The external amplifier uses this signal to drive the motor to the desired position.

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:

$$u(n) = K_p E(n) + K_i \sum E(n) + K_d [E(n') - E(n' - 1)]$$

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  $k_p$ ,  $k_i$ , and  $k_d$  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.

### **Stepper Motor Control**

The MC260 stepper module contains a pulse generator which is used to provide step and direction (or clockwise/counter clockwise) signals to an external stepper motor driver. In addition to auto calibration on power up, the module has an internal feedback loop which accurately maintains the output pulse frequency. The auxiliary encoder inputs of the module can be connected to an optional incremental encoder for motor position verification or closed loop stepper control.

## **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 which 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

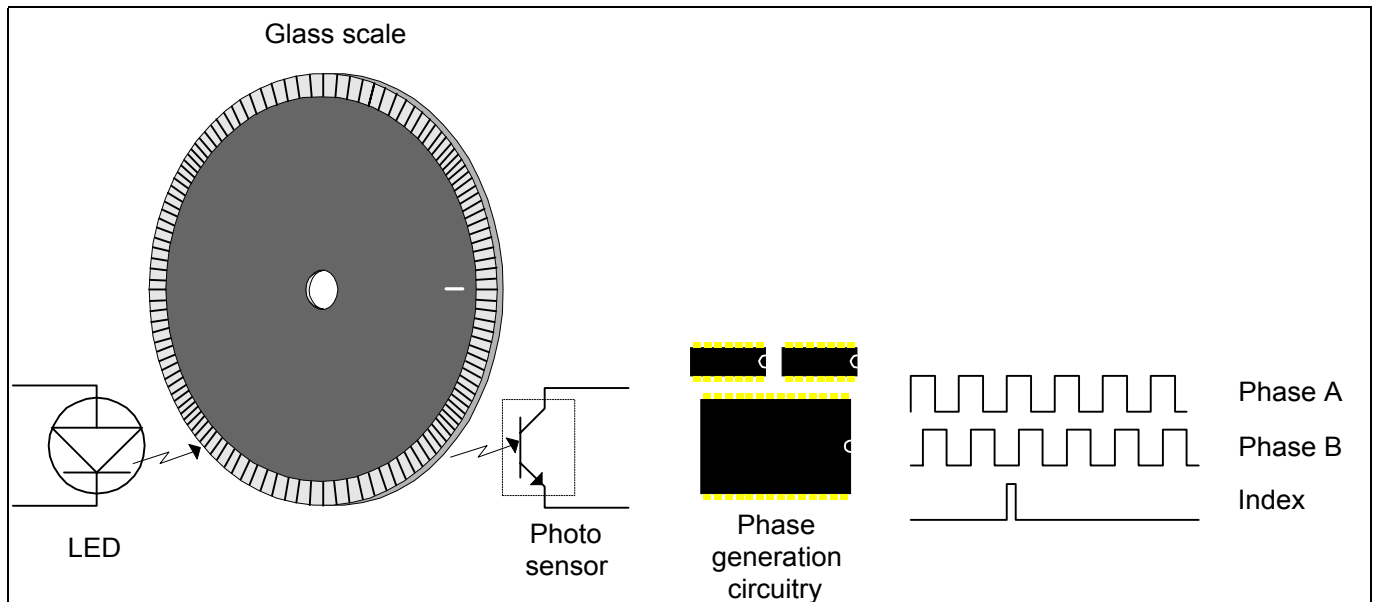
Differential outputs (A+, A-, B+, B-) are recommended for superior noise immunity but the DCX supports either output type

Index or no Index (used for homing the axis)

Differential Index (Z+, Z-) is recommended but the DCX supports single ended Z+ or Z-

+5 volt supply required or +12 volt supply required.

A +5 volt encoder is recommended but the DCX also supports a +12V encoder

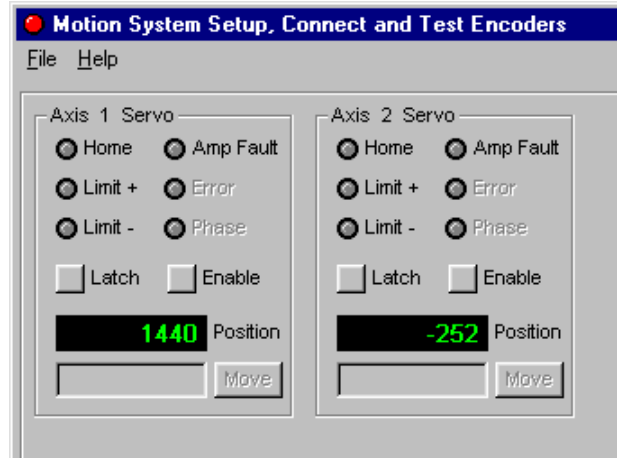
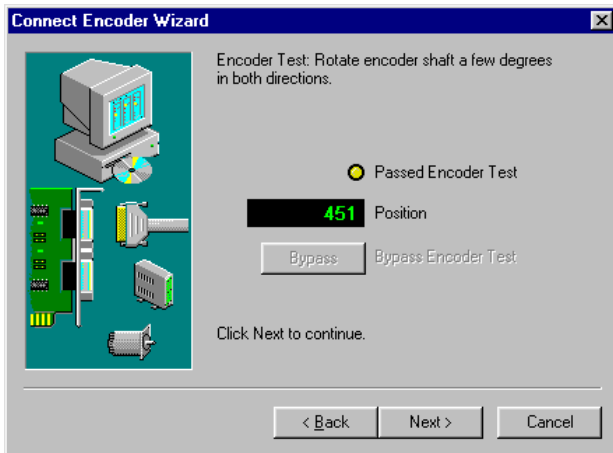


## Encoder Checkout

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.



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.

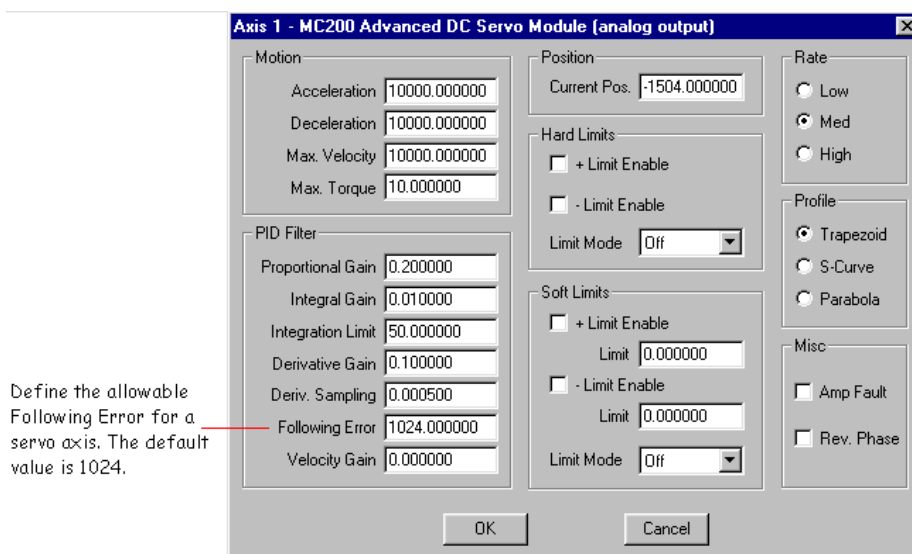
The three conditions that will typically cause a following error are:



- 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 (positive command results in negative motion)

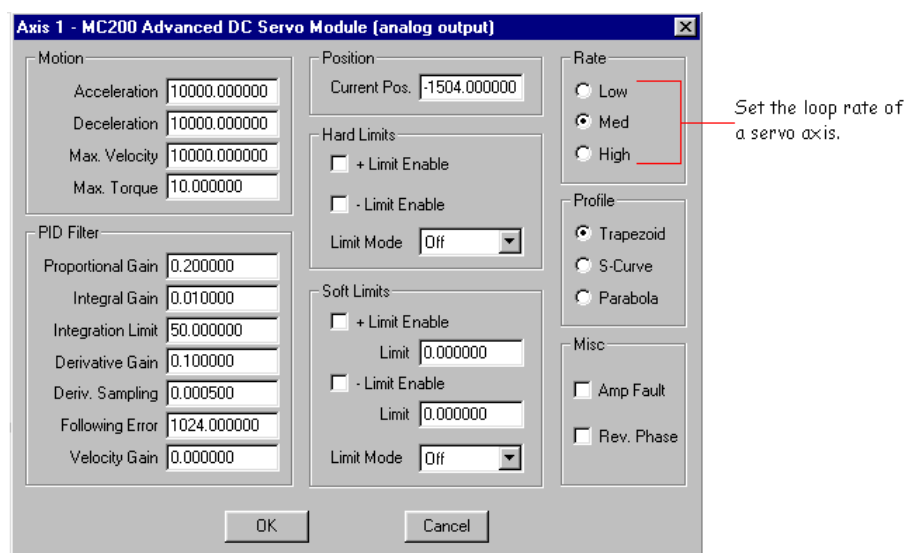
The DCX supports 'hard coded' following error checking. If at anytime the difference between the optimal position and the current position exceeds the user defined 'allowable following error', an error condition will be indicated. The axis will be disabled (Amplifier Enable output turned off, output

command signal set to 0.0V) and the axis status word will indicate that an error has occurred. The **MCEnableAxis()** function is used to clear a following error condition. To disable 'hard coded' following error checking set the allowable following error to zero.



### Selecting the Servo Loop Rate

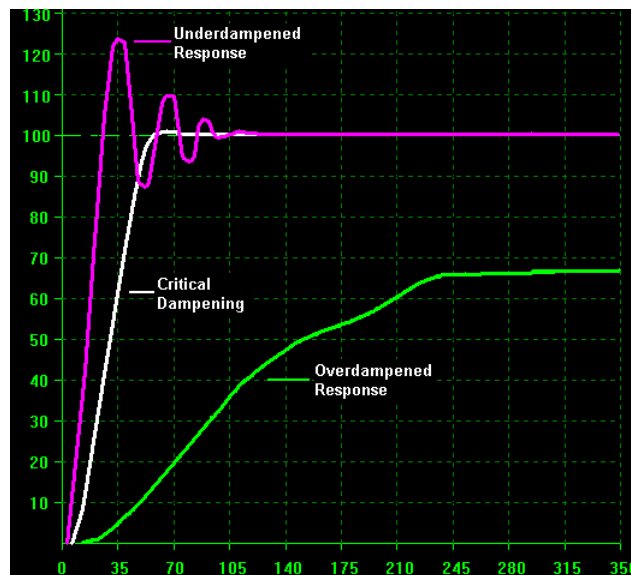
The DCX supports three servo loop rates:



Servo Loop Rate Setting	Description
High	4 KHz servo loop rate (Integral Term 'I' not used)
Medium	2 KHz servo loop rate
Low	1 KHz servo loop rate

## 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:

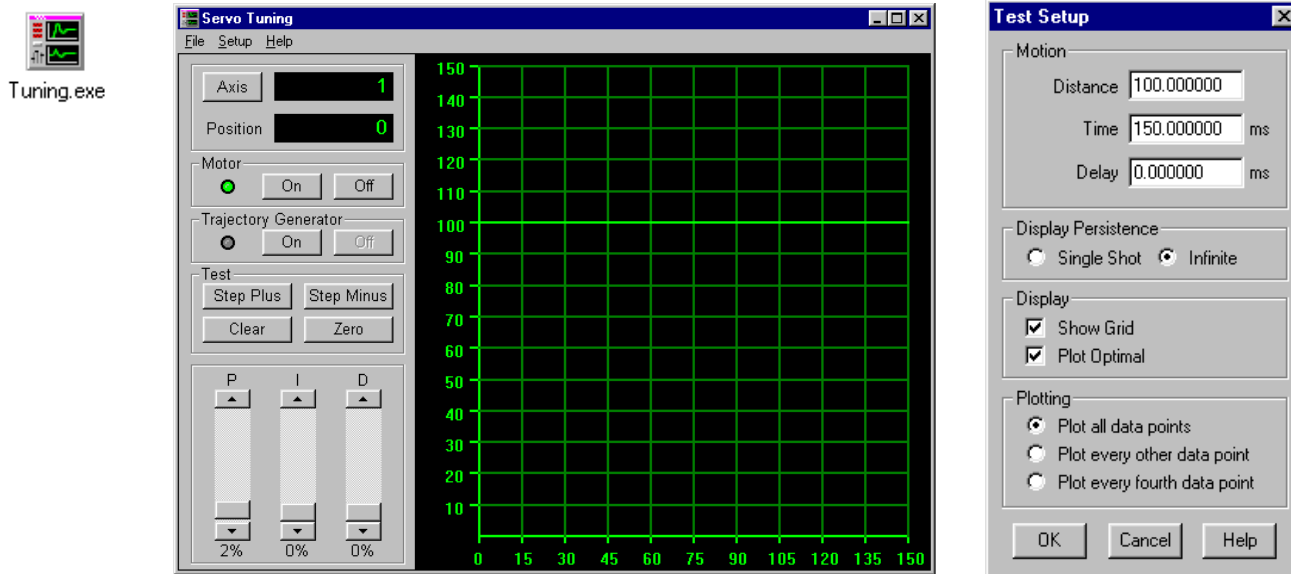
- Very short duration moves (less than 100 msec's)
- Very small following error value (less than 20 encoder counts)

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.

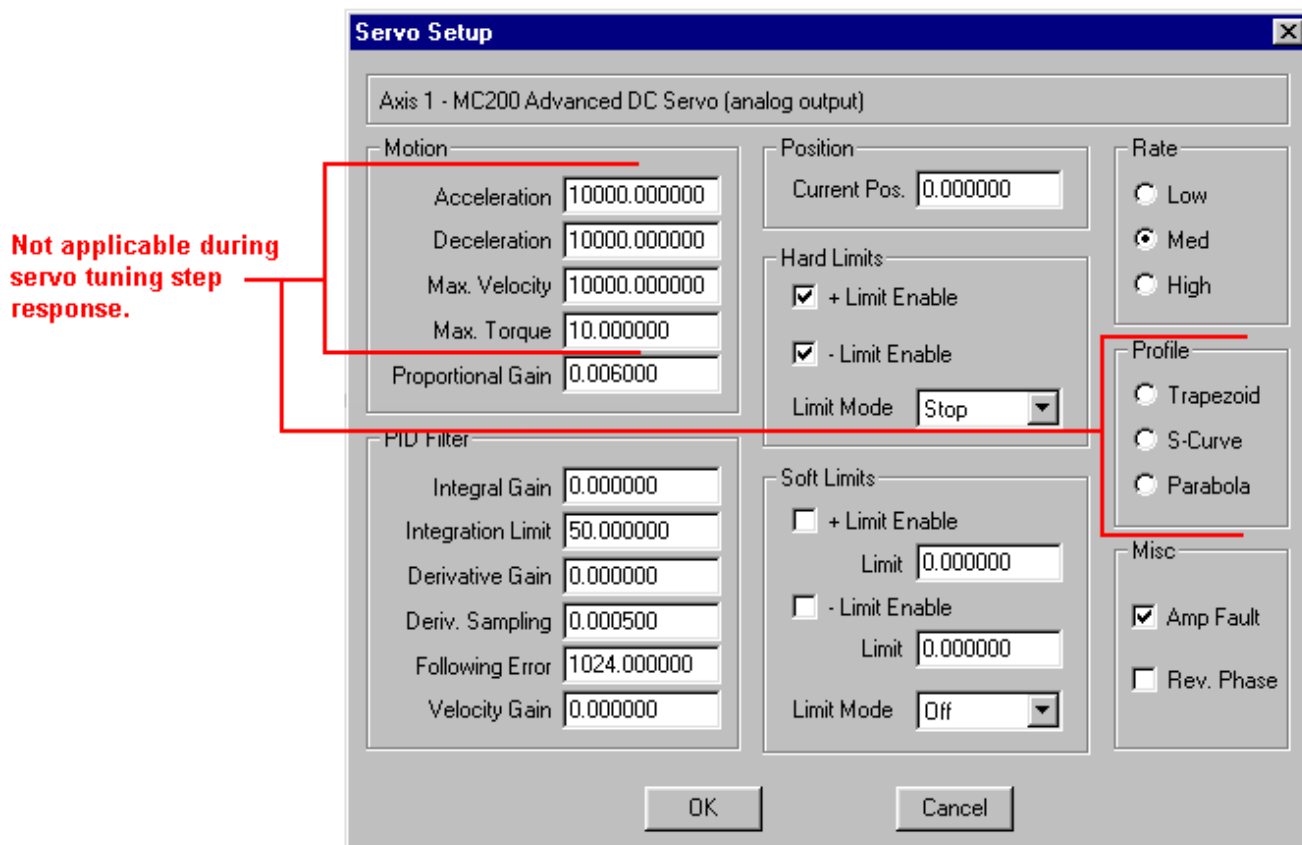


Note – Using a short step response (5 – 20 counts) may result in an unstable system that oscillates during and after a commanded move.

Prior to running PMC's Servo Tuning Utility, WinControl must be closed. Open the Servo Tuning Utility. From the menu bar select **Setup** and then **Test Setup**. Configure the Test Setup dialog as shown (these settings will command a step response of 100 encoder counts displayed over 150 msec's):



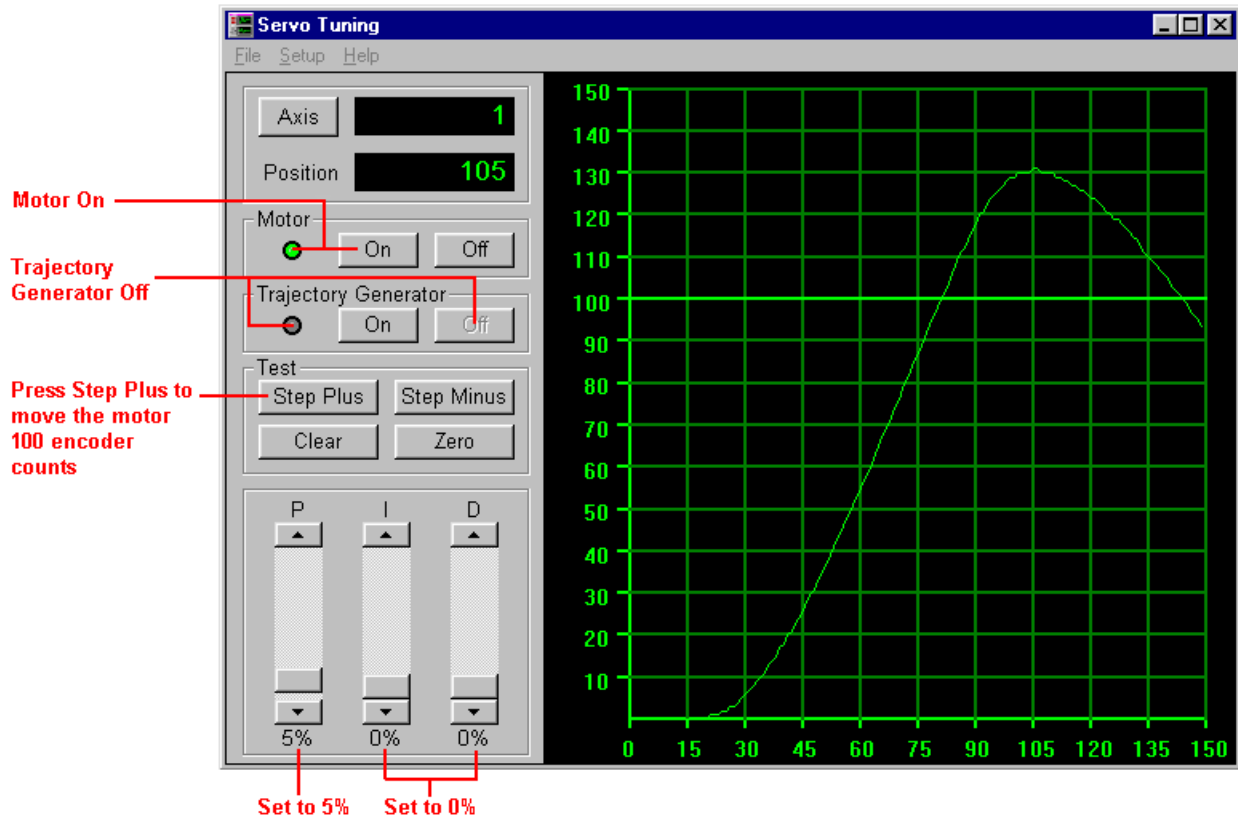
From the menu bar select **Setup** and then **Servo Setup**. Configure the Servo Setup dialog as shown:



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).

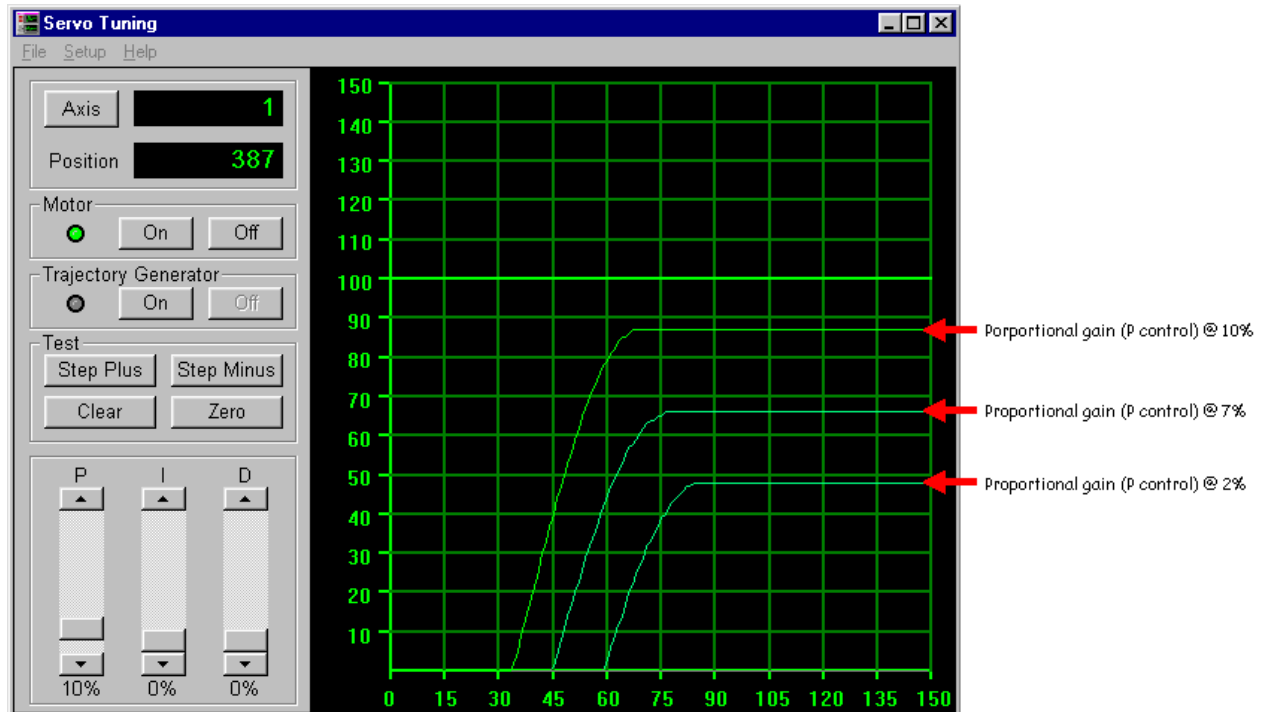
### Setting Proportional Gain

Proportional gain controls the responsiveness of a servo system. Set the 'soft' slide controls for 'I' (Integral gain) and 'D' (Derivative gain) to 0%. Set the slide control for 'P' for 5%. Turn the Motor On. Make sure the Trajectory Generator is **off**. Press the Step Plus button, the motor should move and a position versus time plot will be displayed.

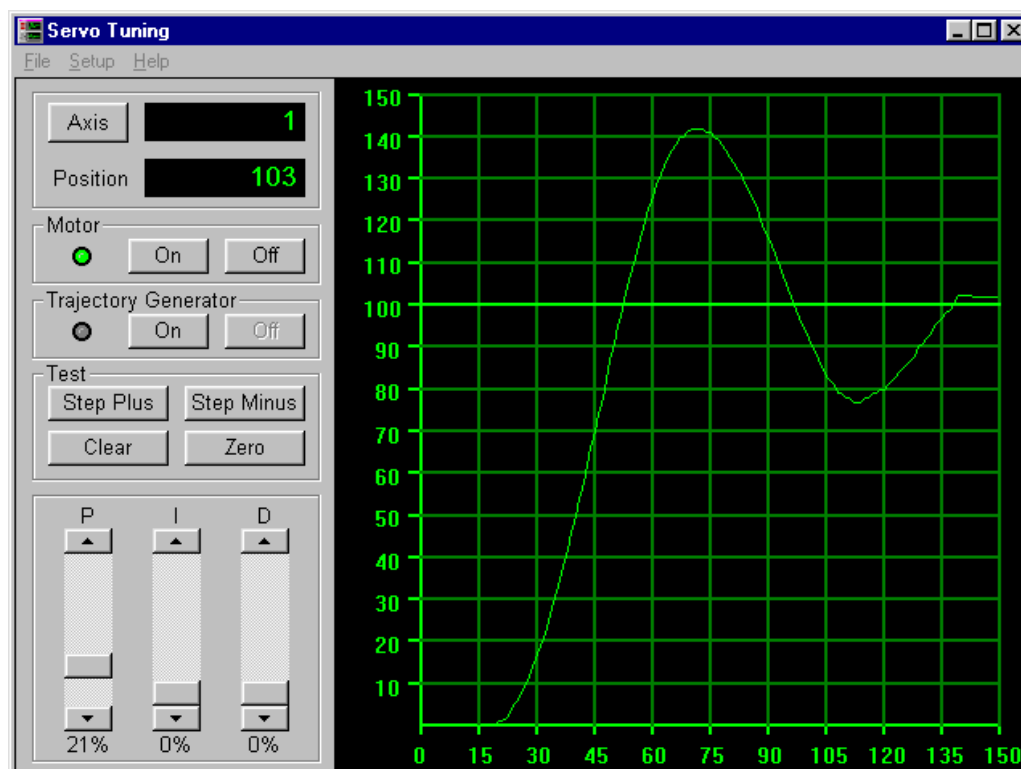


If no plotted position path is shown:

- If the **Motor On LED is still on**, the proportional gain is too low. Increase 'P' by 100%.
- If the **Motor On LED is off**, an error has occurred. The most likely cause is a following error has occurred which would indicate 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. If a position path plot is still not displayed refer to the Troubleshooting chapter of this manual.



Continue to increase the 'P' term until the position versus time plot crosses the target three times (as shown below).

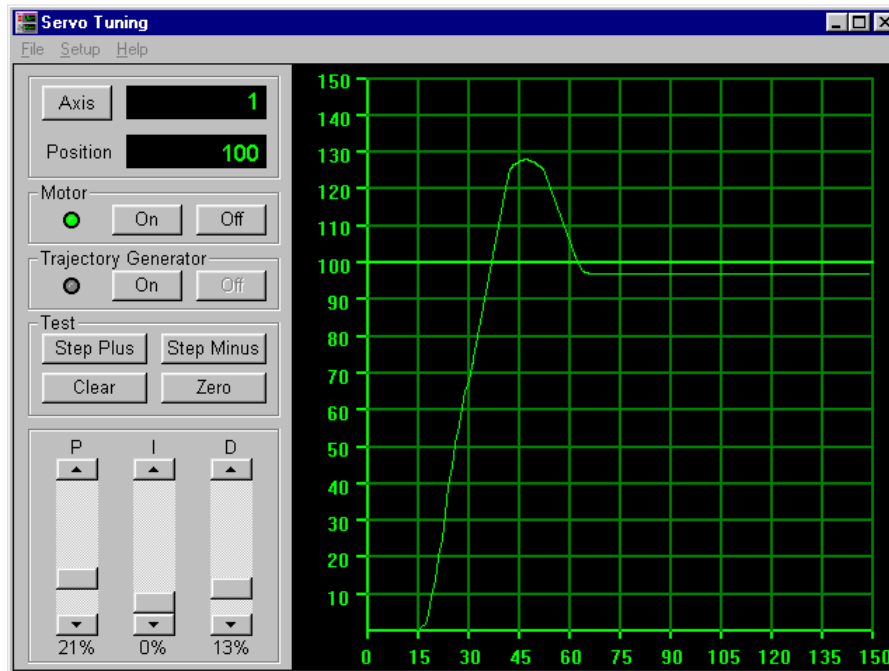


## Setting Derivative Gain

Derivative gain acts as a dampening factor for the servo system. Begin to add Derivative gain by increasing the percentage of the 'D' slide control. The amount of overshoot present in the preceding graphic above will decrease with each new step response. The goal is to:

Limit overshoot to 25% (typical applications)

Limit overshoot to 10% (high performance applications: high velocity and/or short duration moves)



In some high inertia (heavy load with minimal friction) applications, the default setting for the derivative sampling period (.0005 seconds) may be insufficient for achieving acceptable servo performance. If the derivative gain setting is three to five times greater than the proportional gain setting yet the axis is still **significantly** under dampened or is oscillating;

- 1) Double or triple the derivative sampling period setting in the **Servo Setup** dialog.
- 2) Reduce the derivative gain by 75%
- 3) Repeat the step response/gain adjustment process until acceptable performance is achieved

A buzzing or grinding noise is another indication of a system that requires an increase in the derivative sampling period.

## 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:

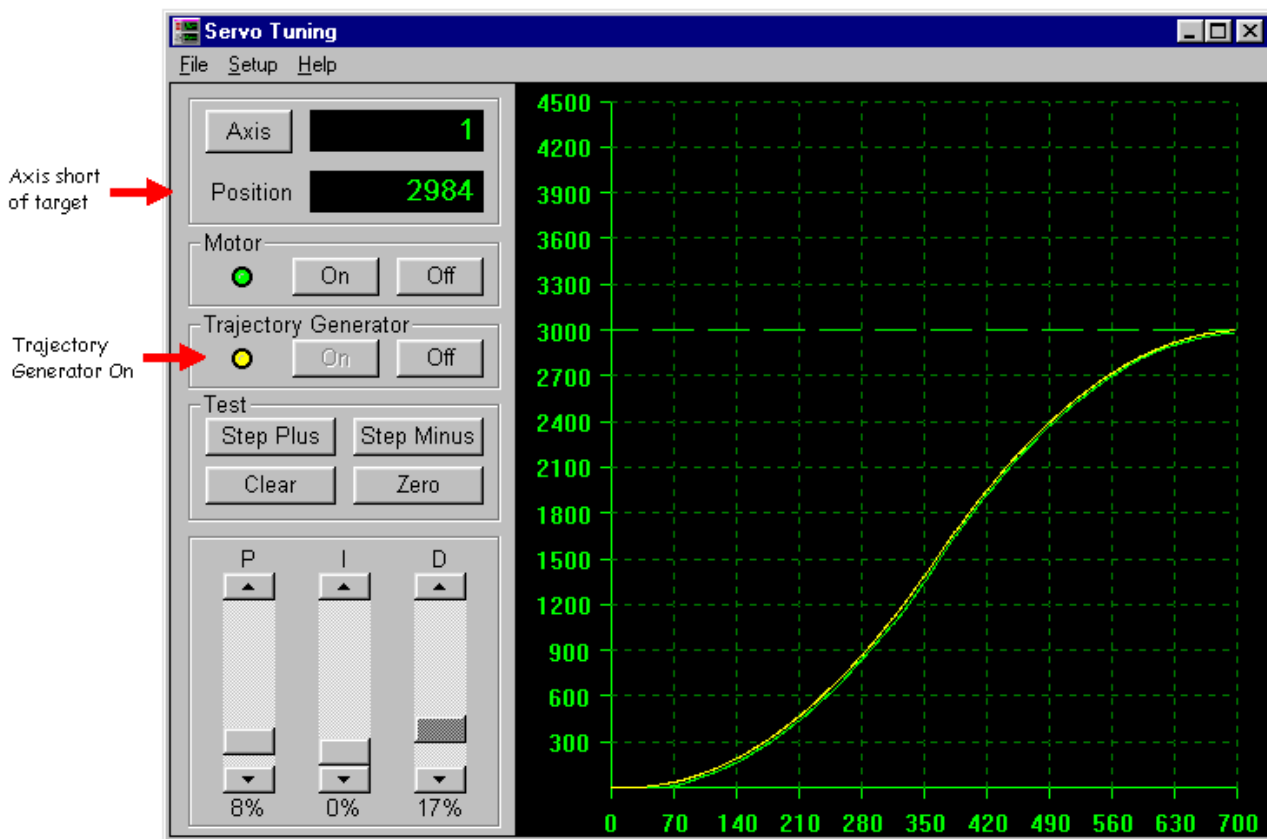


- Enable the trajectory generator.
- Define trajectory parameters (max. velocity, acceleration, and deceleration) in the Servo Setup dialog
- Define a typical application move distance and duration in the Test Setup dialog

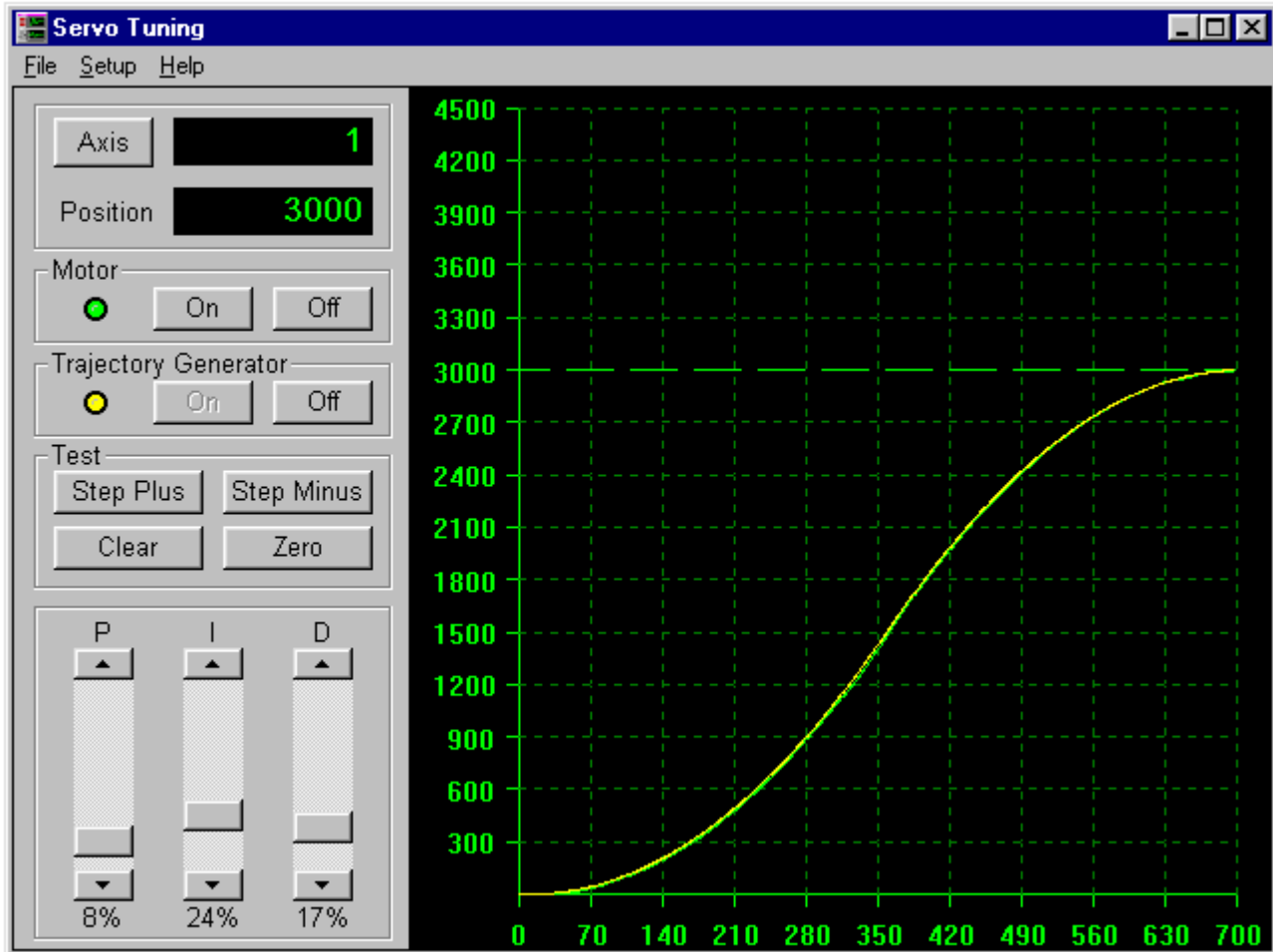
For this example:

- Maximum velocity = 100,000 counts per second
- Acceleration and deceleration = 100,000 counts per second per second
- Move distance = 3,000 counts
- Plot window time = 700 msec's

With the trajectory generator enabled, a step response will cause two plot traces to be displayed by the tuning utility. The green 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, acceleration, and deceleration) defined in the Servo Setup dialog. Prior to setting an integral gain value the system response would be:



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.



If the 'I' control has reached 50% and the axis has not reached the target either:



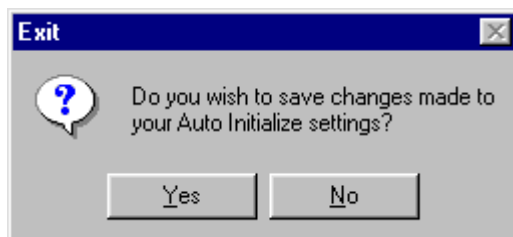
- The Integral Limit is too low, limiting the restoring force that the integral gain can apply. Double the value in the Servo Setup dialog
- The Integral gain slide control range needs to be increased. In the PID setup dialog double the value for the integral gain upper limit

Once the position readout indicates that the axis is at the target execute another move (Step Plus). If the axis stops and settles within one encoder count of the target the servo has been successfully tuned.

If the position readout indicates that the servo is unable to settle, reduce the setting of the integral gain (I term). Execute additional moves until the axis settles at the target.

### 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 Windows folder). Selecting **No** will cause all settings to be discarded.

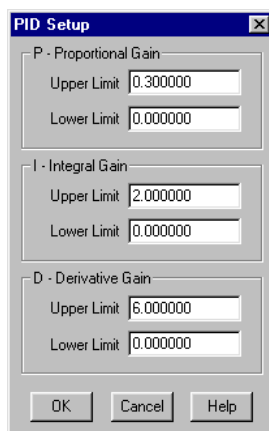


Selecting to save the Auto Initialize settings causes the Servo Tuning utility to call the MCAPI Common Dialog function **MCDLG\_SaveAxis**. All servo parameters (PID, Trajectory, Limits, etc...) will be saved in the dialog

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**).



### Tuning Velocity Mode Amplifier Servo Systems

A velocity mode amplifier incorporates an analog tachometer to provide the feedback for the velocity loop which is closed within the amplifier. The velocity loop is considered the primary or 'inner' loop of this type of servo system. The DCX, which is a position controller, will close the secondary or 'outer'

position loop of the servo system. Combining a velocity mode amplifier with a position loop controller results in what is known as a dual loop system. When this type of system is to be used, it is recommended that the encoder not be directly coupled to the motor. The encoder should be mounted on the external mechanics, as closely coupled as possible to the load or 'end effector'. Typically in a dual loop system, a linear scale (encoder) will be mounted on the slides of each axis.



The most important step of tuning a servo that uses a velocity mode amplifier is to follow the amplifier manufacturers setup instructions **to the letter**. Since the amplifier provides the **primary** servo control, if it is not setup correctly there is no possibility of attaining acceptable servo system performance.

There are significant differences when tuning servo systems that close the velocity loop external to the DCX (position loop) controller. The digital PID filter of the DCX becomes a secondary component in the generation of the output signal that is applied to the velocity mode amplifier. The primary component that the DCX will use to generate the servo command signal is the Feed Forward term.



Feed Forward defines a voltage level output from the DCX, which in turn commands the velocity mode amplifier to rotate the motor at a specific velocity.

Prior to tuning the servo system the velocity feed forward term must be determined. The following example describes how to calculate and set velocity feed forward of a servo axis:

### Setting the Velocity Feed Forward

The main component required to set the velocity feed forward of a DCX servo axis is to determine the output level of the tachometer at a specific motor velocity. For this example, a typical tachometer specification would state:

Output Range	0.0 to +10V
Tach Output @ 1K RPM	1.0 volt

The specification describes a tachometer with an output range of 0 – 10V. The tachometer output ratio is 1.0V per 1,000 RPM's. The resolution of the linear scale encoder is 2000 encoder counts per inch, and the maximum velocity of the axis is 50 inches per second. Note: the servo amplifier may require scaling adjustments for the *RPM/Tachometer voltage output* ratio. The velocity feed forward is calculated as follows:

$$\text{DCX output} = \text{Velocity (encoder counts/sec)} \times \text{Feed forward term (encoder counts/volt/sec.)}$$

$$10 \text{ volts} = 100,000 \text{ counts/sec.} \times \text{Feed forward term (encoder counts} \times \text{volt/sec.)}$$

$$\text{Feed forward} = 10 \text{ volts} / 100,000 \text{ counts per sec.}$$

$$0.0001 = 10 \text{ volts} / 100,000 \text{ counts per sec.}$$

```

1VG0.0001                                ;set velocity gain (velocity feed
                                           ;forward ) with MCCL command

// set velocity gain (velocity feed forward) using MCAPI function
//
MCGetFilterConfig( hCtrlr, iAxis, &Filter );
Filter.VelocityGain    = ( hCtrlr, 1, 0.0001 );
MCSetFilterConfig( hCtrlr, iAxis, &Filter );

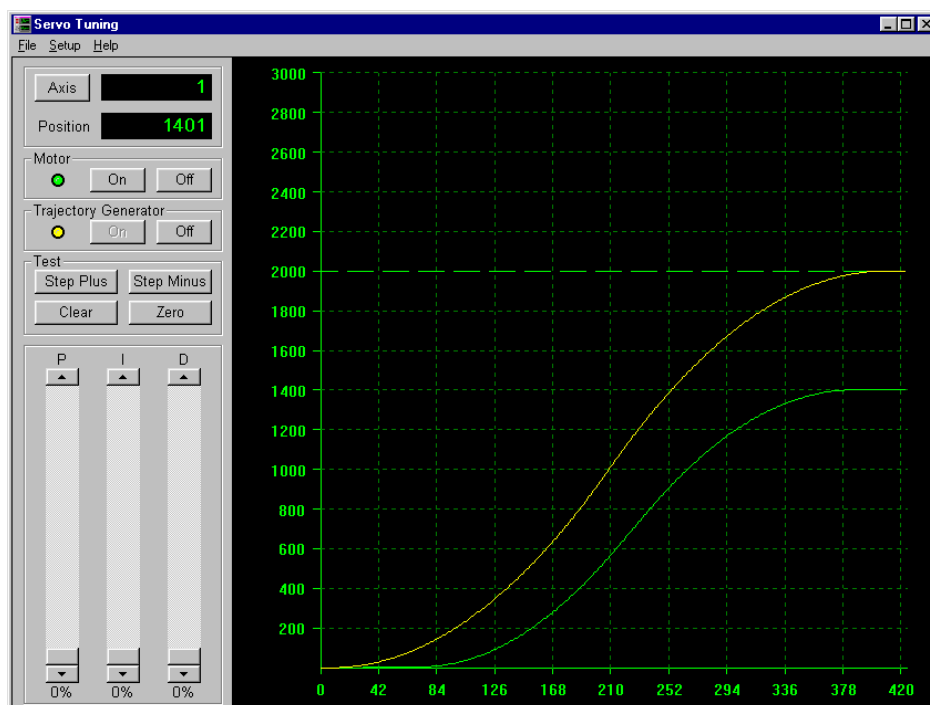
```

## Tuning the Servo

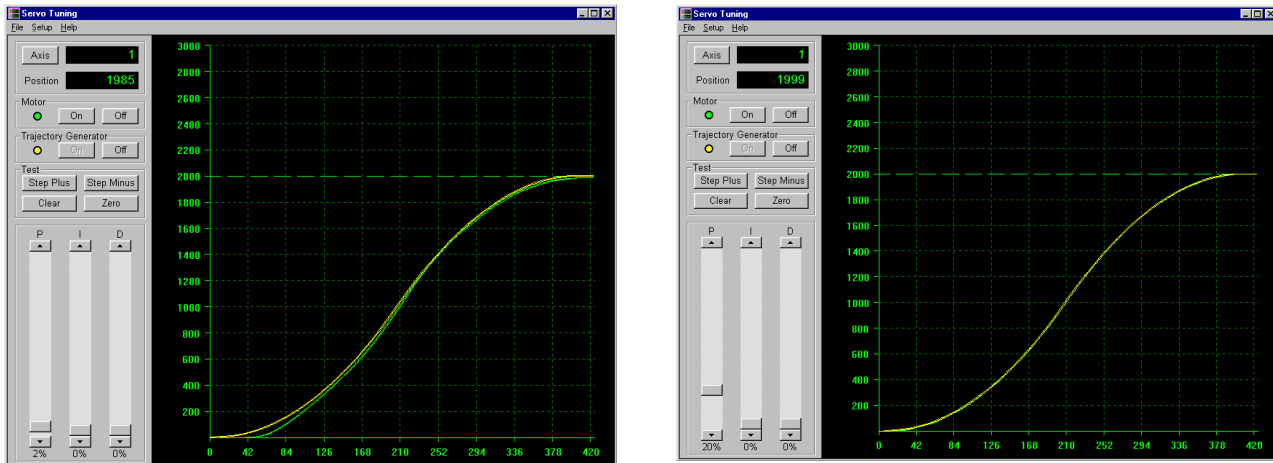
After setting the velocity feed forward (velocity gain) as shown above, open the Servo Tuning Utility. Configure the utility as follows:

- 1) From the Setup menu, select Servo Setup and define the trajectory parameters (velocity, acceleration, and deceleration) to match the application requirements.
- 2) From the Test Setup menu define a typical application move distance and duration. For this example, the move distance is set to 2000 encoder counts. The move duration is set to 420 milliseconds.
- 3) Set the Proportional (P), Integral (I), and Derivative (D) slide controls to 0%.
- 4) Turn on the Trajectory generator
- 5) Turn the motor on
- 6) Press the Step Plus pushbutton

A response similar to the following graphic should be observed:



Increase the 'P' term 1-2 % at a time until the position display indicates that the axis is within +/- 2 counts of the target.

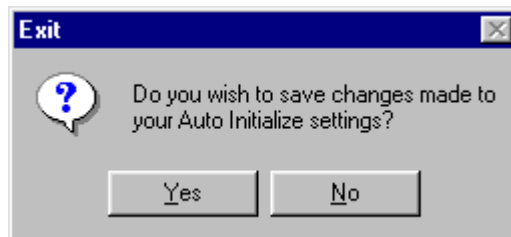


Increase the "I" term 1% at a time until the axis repeatedly positions to the target. If increasing the Integral setting causes the axis becomes unstable:

- 1) Reduce the Integral Limit setting (Setup – Servo Setup)
- 2) Reduce the scale of the 'I' term slide control (Setup – PID setup)

### Saving the Tuning Parameters

When servo tuning is complete, closing the tuning utility will prompt this message about saving the Auto Initialize setting, selecting **Yes** will store all settings for all installed axes. Selecting **No** will cause all settings to be discarded.



### Acceleration and Deceleration Feed Forward

For most applications velocity feed forward is sufficient for accurately positioning the axis. However for applications that require a very high rate of change, acceleration and deceleration gain must be used to reduce the following error at the beginning and end of a move.

Acceleration and deceleration feed forward values are calculated using a similar algorithm as used for velocity gain. The one difference is the velocity is expressed as encoder counts per second, while acceleration and deceleration are expressed as encoder counts per second per second.

$$\text{DCX output} = \text{Accel./Decel. (encoder counts/sec/sec.)} \times \text{Feed forward term (encoder counts * volt/sec./sec.)}$$



Acceleration and deceleration feed forward values should be set prior to using the Servo Tuning Utility to set the proportional and integral gain.

### Systems with Electrical or Mechanical Dead Band

Some servo systems may demonstrate significant dead band due to friction, sticktion, or insufficient amplifier drive power. This will typically be indicated when the output command to the servo is relatively high but the axis does not move.

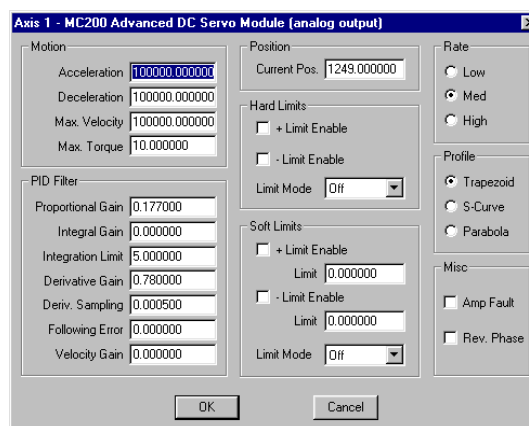
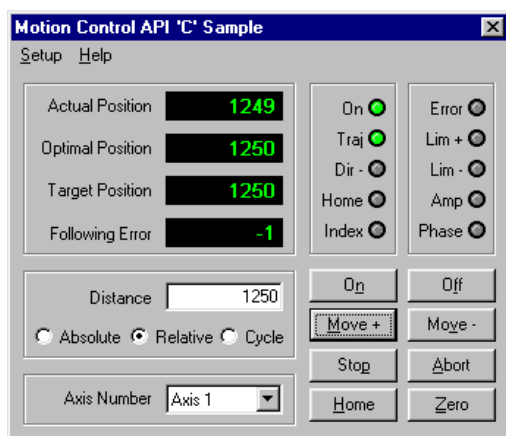
Systems of this type can be very difficult to 'tune'. To overcome the limitations of the system and get the axis moving, the proportional gain would need to be set very high. This will tend to make the system become unstable, causing the axis to 'oscillate' at the end of a move. The **define an Output dead band** (aODn) command is used to compensate for the electrical and or mechanical dead band in a system by modifying the calculated output signal, allowing the module to simulate a 'frictionless' system. The deadband value will be added to a positive output and subtracted from a negative output.

### Programming an Output Offset

Both the MC200 and MC210 servo modules have output offset adjustment potentiometers for manually setting the zero point of the servo command output. The Output Offset command allows the user to enter a programmable output offset ranging from -10V to +10V (MC200) and 0 to 100% duty cycle (MC210).

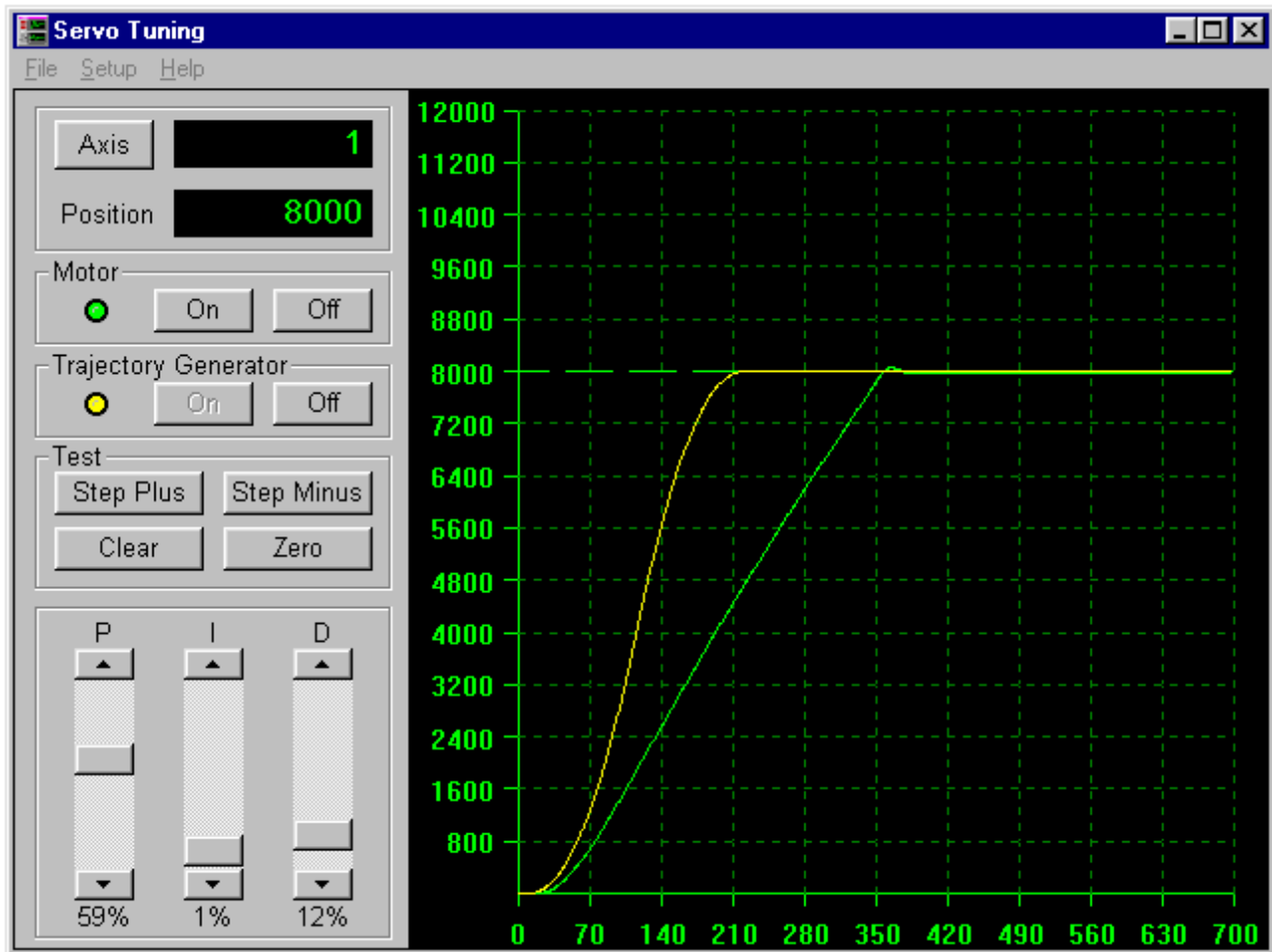
### Moving an Axis

Once the servo is tuned, the axis is ready to perform velocity profile moves. The demo programs (Cwdemo is shown below) allow the user to execute absolute, relative, and cycle move sequences, monitor position and status of the axis. By selecting **Configure Axis** from the **Setup menu** the user can; set velocity parameters (maximum velocity, acceleration, and deceleration), set velocity profile (Trapezoidal, S curve, or Parabolic), and enable motion limits.



By turning on the Trajectory Generator while in the Servo Tuning Utility, its plotting capabilities can be used to display the performance of the axis during a velocity profile move. In this mode two sets of

points are plotted. The yellow trace is the optimal position (as calculated by the DCX), the green trace is the actual position of the axis. The difference between the two plots is the following error.





## DCX Stepper Basics

The DCX motion control system supports both open loop and closed loop stepper motion.

### Open Loop Stepper Motion

As long as the performance of the selected stepper motor matches the applications requirements for:

- Torque vs. load
- Maximum velocity
- Step size/positioning resolution
- Required rate of change

the motion control system will not require position or velocity feedback to accurately position an axis. Commanding motion of a motor with no position or velocity feedback is known as 'Open Loop'. To successfully complete the commanded move, the DCX controller counts each step pulse issued to the stepper motor driver. When the position of an axis is queried (by issuing the function **MCGetPosition** () or **MCCL Tell Position** (aTP) command), the number of pulses issued to the stepper driver is reported. Since there is no position (or velocity) feedback there is no need to 'tune' the axis. However, the axis module must be configured (Trajectory parameters, Velocity Profile, Limits etc...). Please refer to the following stepper setup dialog:

**Axis 1 - MC260 Advanced Stepper Module**

**Motion**

Acceleration: 100000.000000  
Deceleration: 100000.000000  
Max. Velocity: 125000.000000  
Min. Velocity: 250.000000

**Position**

Current: 0.000000

**Hard Limits**

☒ + Limit Enable  
☒ - Limit Enable  
Limit Mode: Stop

**Soft Limits**

☐ + Limit Enable  
Limit: 0.000000  
☐ - Limit Enable  
Limit: 0.000000  
Limit Mode: Off

**Rate**

☐ Low ☒ Med ☐ High

**Profile**

☒ Trapezoid  
☐ S-Curve  
☐ Parabola

**Miscellaneous**

☒ Half Step ☒ Low Current

OK Cancel

**Define the velocity parameters**

**Set the Step range:**  
Low = 15 - 19.5K steps/sec.  
Medium = 125 - 156K steps/sec.  
High = 1K - 1M steps/sec.

**Define the velocity profile**

**Define the Limit Mode**



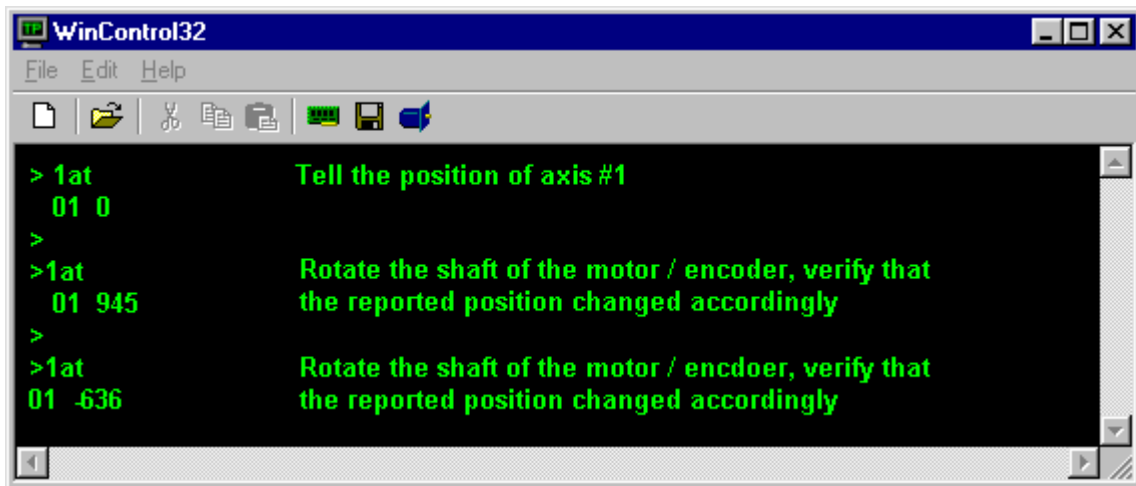
The minimum velocity of a stepper axis must be set to a non zero value. The default value is 1,000 steps per second. The recommended setting of the minimum velocity is from 1% to 10% of the maximum velocity.

## Closed Loop Steppers

The advancements in stepper motor/micro stepping driver technology have allowed many machine builders to maintain 'servo like' performance while reducing costs by moving to closed loop stepper systems. While closed loop steppers are still be susceptible to 'stalling', they are not plagued with the familiar open loop stepper system problem of loosing steps due to encountering friction (mechanical binding) or system resonance.

For high accuracy stepper applications, the DCX supports closed loop control of stepper motors using quadrature incremental encoders for position feedback. The stepper axis will be controlled as if it is a closed loop servo, the quantity and frequency of step pulses applied to the stepper driver is based on the trajectory parameters of the move and the position error of the axis. Prior to addressing the closed loop operation, the stepper axis must be installed and configured as described in the previous section (Open Loop Stepper Motion).

Connect the stepper's encoder to the module, the DCX should report the position of the encoder each time the **report the position of the Auxiliary encoder (AT)** command is issued. If the encoder is manually turned in either direction, the position reported should increment or decrement accordingly.



The current release of the Motion Control API (2.20.0000) does not provide high level function calls for configuring the DCX controller for closed loop stepper motion.



Any of the MCCL commands used in the following description can be issued to the controller via the MCAPI OEM low level function calls ***pmccmdex()*** and ***pmcrpyex()***.

Future releases of the MCAPI will resolve this lack of support.

To enable Closed loop stepper mode issue the Input Mode (aIMn) command with parameter n equal to 1. The MCAPI functions **MCGetPosition( )** and **MCGetAuxEncPosEx( )** will now report the same value, which is the number of encoder counts decoded by the DCX since the axis was last homed.

Configuring an axis for closed loop stepper motion requires the user to define the velocity gain of the axis. The algorithm for calculating the velocity gain of a closed loop stepper axis:

$$\text{Closed loop Velocity Gain} = \text{MC260 Calibration Constant} \times \frac{\text{Step motor pulses per rotation}}{\text{Encoder counts per rotation}}$$

The MC260 calibration constant is a value generated by self test diagnostics upon power/reset. This calibration value is defined as the default velocity gain of the axis and can be retrieved using the **MCGetFilterConfig( )** function.

### Closed loop stepper example

An incremental encoder coupled to the shaft of a stepper motor. The encoder has 4000 counts per rotation (1000 lines). The stepper motor is connected to a micro stepping driver configured for 50,000 steps per motor rotation. The required maximum step rate of the axis is 625,000 or (750 RPM). This step rate requires the axis to be configured for High Speed step range using the function **MCSetMotionConfig( )**. After the step rate range is defined, the calibration constant is retrieved using the Tell Konstant (aTKn) command:

```
1HS
1TK.3                                ;report the calibration constant of
                                      ;axis #1
```

the controller responds with the value:

```
01      0.746
```

The Velocity Gain is calculated as follows:

$$\text{Closed loop Velocity Gain} = \text{MC260 Calibration Constant} \times \frac{\text{Step motor pulses per rotation}}{\text{Encoder counts per rotation}}$$

$$\text{Closed loop Velocity Gain} = 0.746 \times \frac{50,000}{4,000}$$

$$\text{Closed loop Velocity Gain} = 0.746 \times 12.5$$

$$\text{Closed loop Velocity Gain} = 9.325$$

**Filter.VelocityGain = 9.325**

For some closed loop stepper applications, setting the velocity gain of the axis is the only 'tuning' the axis will require. Most applications will require that the proportional gain be used to:

Minimize the following error while moving

Eliminate slow speed slewing of the axis near the end of the move.

Currently there are no MCAPI software utilities available for tuning the proportional gain of a closed loop servo. The following MCCL command sequences is used to determine the correct setting for proportional gain (**MCSetGain( )**).

```
1SG1                                ;set the proportional gain to a
                                   ;minimum value
1MR100000                          ;execute a 2 second move
WA.5,1TF,WA.25,1TF,WA.25,1TF,WA.25,1TF ;report the following error

01 -62                             ;reported following error @ time .5 sec
01 -195                            ;reported following error @ time .75 sec
01 -369                            ;reported following error @ time 1.0 sec
01 -542                            ;reported following error @ time 1.25 sec
```

increase the proportional gain until the following error value stops increasing

```
1SG10                              ;increase proportional gain to 10

01 -57                             ;reported following error @ time .5 sec
01 -168                            ;reported following error @ time .75 sec
01 -287                            ;reported following error @ time 1.0 sec
01 -375                            ;reported following error @ time 1.25 sec
```

```
1SG50                              ;increase proportional gain to 50

01 -39                             ;reported following error @ time .5 sec
01 -86                             ;reported following error @ time .75 sec
01 -117                            ;reported following error @ time 1.0 sec
01 -124                            ;reported following error @ time 1.25 sec
```

```
1SG100                             ;increase proportional gain to 50

01 -24                             ;reported following error @ time .5 sec
01 -55                             ;reported following error @ time .75 sec
01 -62                             ;reported following error @ time 1.0 sec
01 -62                             ;reported following error @ time 1.25 sec
```

A proportional gain setting of 100 will result in acceptable stepper performance. The following command sequence will configure and operate stepper axis number one in closed loop mode:

```
1HS                                ;Select Stepper speed range
1TK.3                             ;Report the current calibrated
                                   ;velocity gain
01 0.746                          ;Controller will report the MC260
                                   ;calibrated gain
1VG9.25                           ;Set the velocity gain (1VG9.25)
                                   ;Multiply the calibrated gain by the
                                   ;ratio of step pulses per rotation to
                                   ;encoder counts per rotation.

1IM1                              ;Enable closed loop stepper mode
1AH                              ;Zero the position of the encoder
```

---

1MN	;Turn on the axis. The target and optimal ;positions will be initialized to the current ;position of the encoder.
1SV75000,1SA100000,1DS100000	;Set maximum velocity, acceleration, and ;deceleration in units of encoder counts
1MV200	;Set the minimum velocity so the motor ;reaches the target position with some ;non-zero velocity. Typically between 5% - ;10% of the maximum velocity. ; <b>Warning:</b> The minimum velocity must be less ;than the maximum velocity.
1SE500	;Set the allowable maximum following error ;in encoder counts.
1SG100,1SD0,1SI0	;Set the proportional gain, derivative gain, ;integral gain, and integration limit for ;best motor speed stability.
1MR200000	;Issue moves in units of encoder counts. ;Watch the axis error LED's on the DCX ;motherboard for indication a motor ;error which can be caused by the axis ;exceeding the maximum following error ;or being reversed phased.

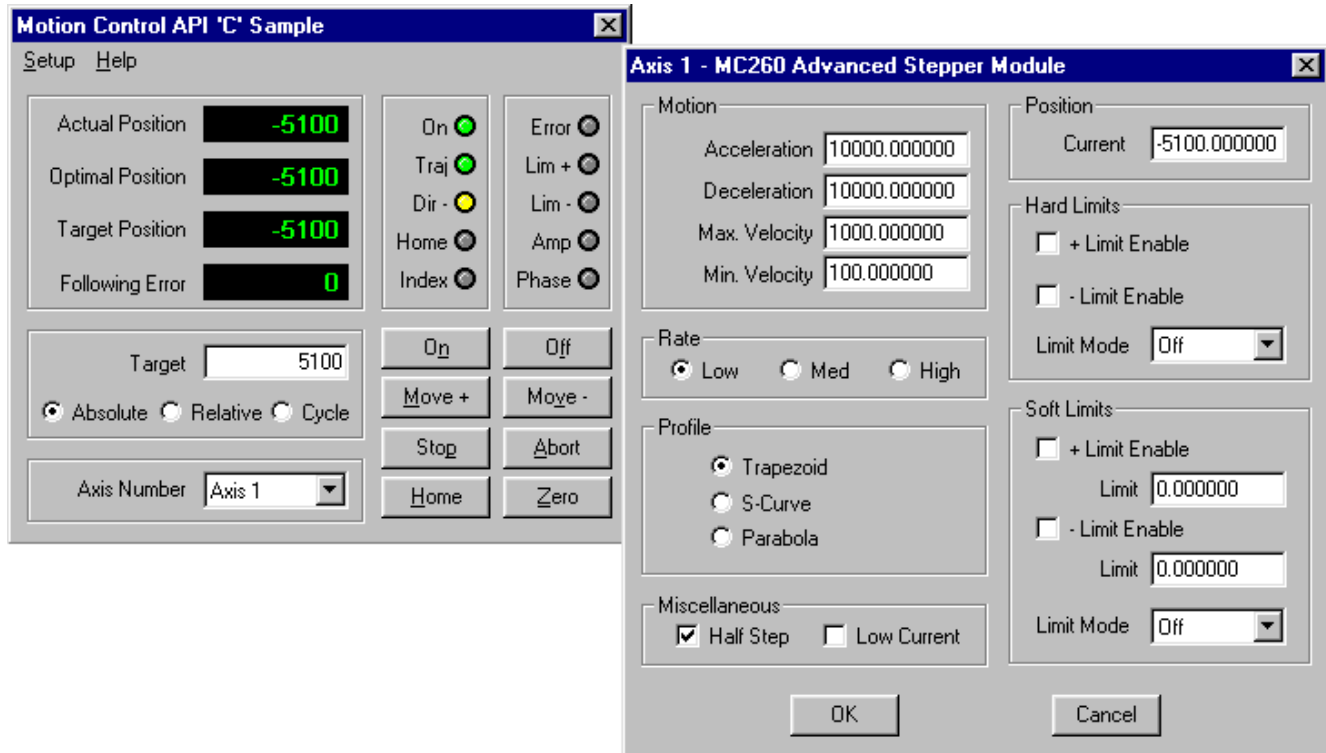
Be aware that if the stepper is reverse phased, issuing a move command will cause the motor to 'take off' in the wrong direction at full torque or speed. In this case, once the position error exceeds the value entered using the Stop on Error command (default = 1024) a motor error will occur and the axis will stop. If this happens, the phasing can be changed by issuing the PHase (aPHn) command to the axis with a parameter of 1, or reverse the encoder phase A and B connections to the MC360 module. If the motor is properly phased, it should resist movement away from its current position.



When the **MCWaitForStop()** function is issued to a closed loop stepper, additional MCAPI function execution will be delayed until the motor has reached the target position of the move.

## Moving Motors with PMC demo's

After defining the step output mode and the step range the axis is ready to execute motion. The demo programs (CWDemo32 is shown below) allow the user to execute absolute, relative, and cycle move sequences, monitor position and status of the axis. By selecting **Configure Axis** from the **Setup menu** the user can; set velocity parameters (maximum velocity, acceleration, and deceleration), set velocity profile (Trapezoidal, S curve, or Parabolic), and enable motion limits.



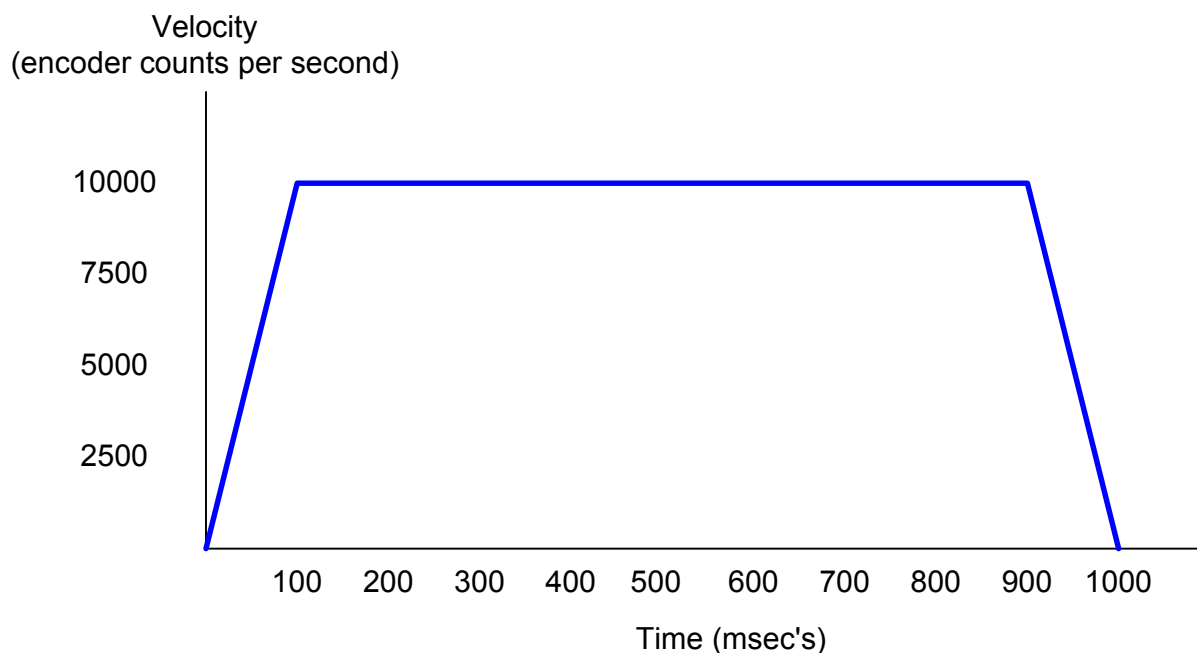
## Defining the Characteristics of a Move

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

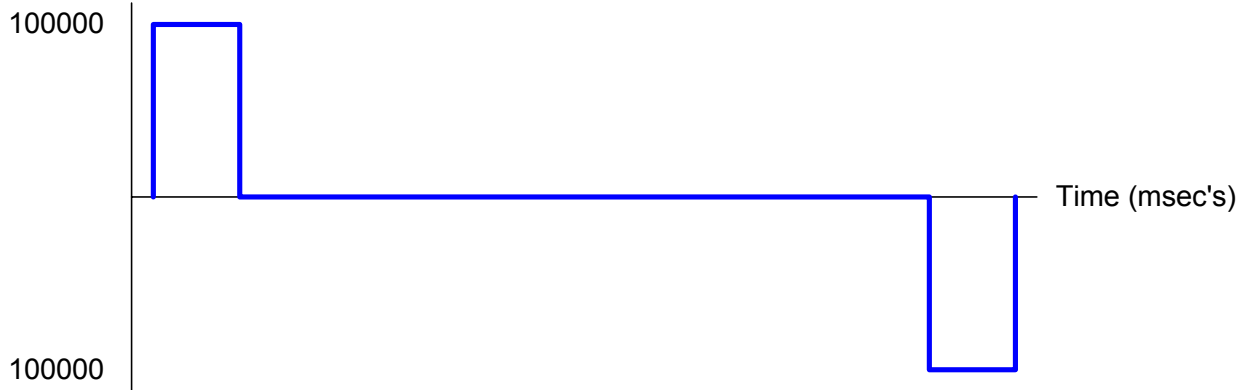
```
// Set axis 1 maximum velocity
// Set axis 1 acceleration
// Set axis 1 deceleration
// Set profile as Trapezoidal
// Set Position mode
// Set target (10000), begin move

MCSetVelocity( hCtrlr, 1, 10000.0 );
MCSetAcceleration( hCtrlr, 1, 100000.0 );
MCSetVelocity( hCtrlr, 1, 100000.0 );
MCSetProfile( hCtrlr, 1, MC_PROF_TRAPEZOID );
MCSetOperatingMode( hCtrlr, 1, 0, MC_MODE_POSITION );
MCMoveRelative( hCtrlr, 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, thereby reaching the maximum velocity (10,000 counts per second) in 100 msec's. The resulting velocity and acceleration profiles follow:



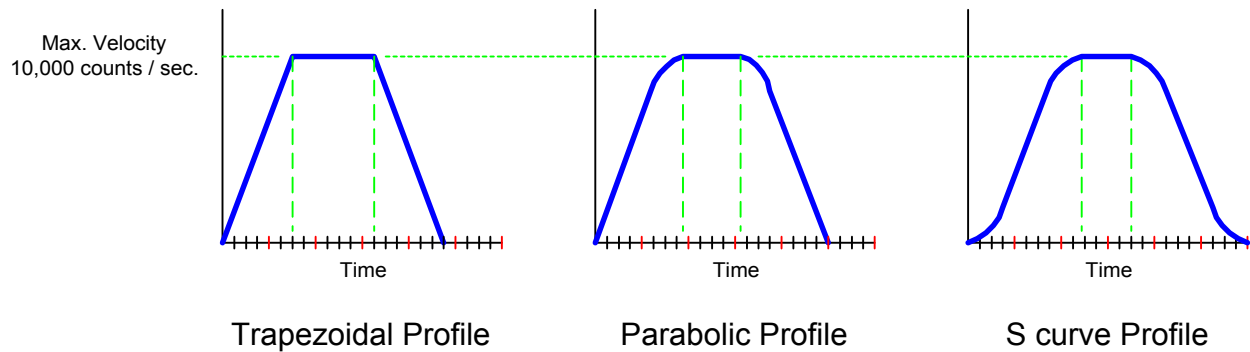
Acceleration / Deceleration  
(encoder counts per sec / sec)



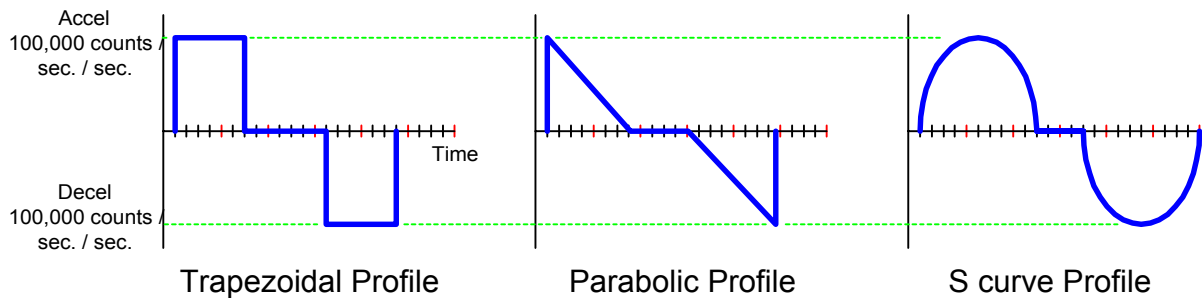
## Velocity Profiles

The user can select one of three different velocity profiles that the DCX will then use to calculate the trajectory of a move.

DCX Velocity Profiles



DCX Accel / Decel Profiles





**Trapezoidal Profile** – (servo & steppers) `MCSetProfile( hCtrlr, 1, MC_PROF_TRAPEZOID );`  
 Shortest time to target when using the same trajectory parameters  
 Profile most likely to result ‘jerk’ and/or oscillation  
 Supports ‘on the fly’ target changes

**Parabolic Profile** – (stepper only) `MCSetProfile( hCtrlr, 1, MC_PROF_PARABOLIC );`  
 Slow ‘roll off’ minimizes lost steps at high velocity  
 Initial linear rate of change eliminates ‘cogging’  
 On the fly changes will cause the axis to first decelerate to a stop

**S curve Profile** – (servo only) `MCSetProfile( hCtrlr, 1, MC_PROF_SCURVE );`  
 ‘True sine’ rate of change effectively eliminates ‘jerk’ and/or oscillation  
 Longest time to target when using the same trajectory parameters  
 On the fly changes will cause the axis to first decelerate to a stop

## Point to Point Motion

To perform point to point motion of a servo or stepper motor, the following steps are required:

```
// Enable the axis
// Enable Position mode
// Define the velocity profile (trapezoidal, S curve, or parabolic)
// define maximum velocity
// define acceleration
// define deceleration
// execute the move

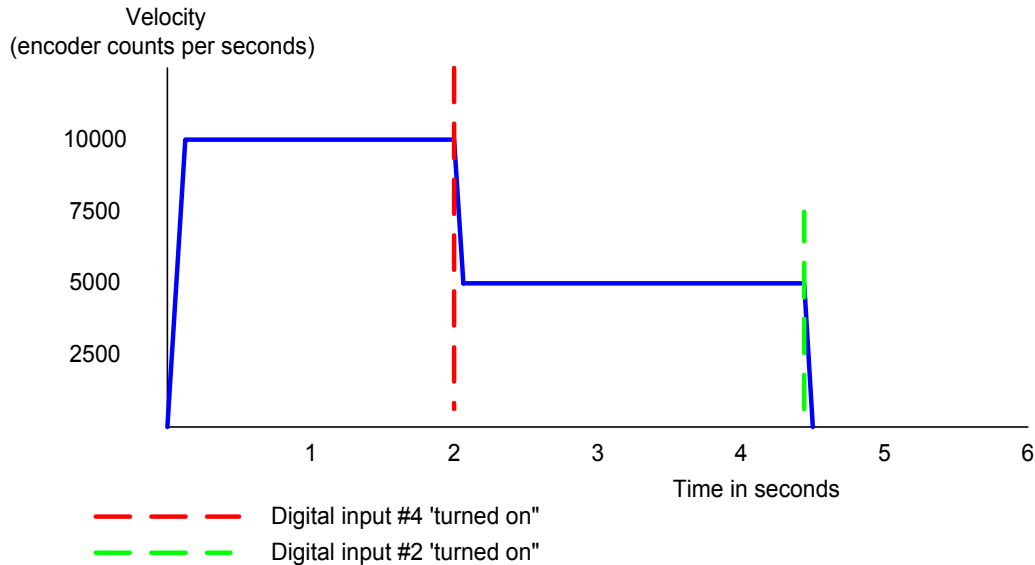
MCEnableAxis( hCtrlr, 1, TRUE );
MCSetOperatingMode( hCtrlr, 1, 0, MC_MODE_POSITION );
MCSetProfile( hCtrlr, 1, MC_PROF_TRAPEZOIDAL );
MCSetVelocity( hCtrlr, 1, 10000.0 );
MCSetAcceleration( hCtrlr, 1, 25000.0 );
MCSetDeceleration( hCtrlr, 1, 50000.0 );
MCMoveRelative( hCtrlr, 1, 122.5 );
```

## Constant Velocity Motion

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

```
// Enable the axis
// Enable Velocity mode
// Define the velocity profile (trapezoidal, S curve, or parabolic)
// define maximum velocity
// define acceleration
// define 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( hCtrlr, 1, TRUE );
MCSetOperatingMode( hCtrlr, 1, 0, MC_MODE_VELOCITY );
MCSetProfile( hCtrlr, 1, MC_PROF_TRAPEZOIDAL );
MCSetVelocity( hCtrlr, 1, 10000.0 );
MCSetAcceleration( hCtrlr, 1, 100000.0 );
MCSetDeceleration( hCtrlr, 1, 100000.0 );
MCSetDirection( hCtrlr, 1, POSITIVE );
MCGo( hCtrlr, 3 );
MCWait For DigitalIO( hCtrlr, 4, TRUE );
MCSetVelocity( hCtrlr, 1, 5000.0 );
MCWait For DigitalIO( hCtrlr, 2, TRUE );
MCStop( hCtrlr, 1 );
```



## Contour Motion (arcs and lines)

The DCX supports Linear Interpolated motion with any combination of two to six axes and Circular Contouring on as many as three groups of two axes. Executing a multi axis contour move requires:

- Turn the axes on
- Define the axes in the contour group and the controlling axis
- Define the trajectory (Vector Velocity, Vector Acceleration and Vector Deceleration)
- Define the type of contour move (Linear, Circular, user defined) and the move targets
- Loading the Contour Buffer for Continuous Path Contouring

### Defining the contour group

The **MCSetOperatingMode()** command is used to define the axes in a contour group. Issue this command to each of the axes in the contour group. The parameter *wMaster* should be set to the lowest axis number of the servo or stepper motor that will be moving on the contour. This axis will then be defined as the 'controlling' axis for the contour group. The following example configures axis 1, 2, and 3 for contour motion with axis #1 defined as the controlling axis.

```
MCSetOperatingMode( hCtrlr, 1, 1, MC_MODE_CONTOUR );
MCSetOperatingMode( hCtrlr, 2, 1, MC_MODE_CONTOUR );
MCSetOperatingMode( hCtrlr, 3, 1, MC_MODE_CONTOUR );
```

### Define the trajectory parameters

The **MCGetContourConfig()**, **MCSetContourConfig()**, and **MCContour** data structure are used to define the trajectory parameters of a contour motion. The default units of the vector velocity are encoder counts or steps per second. The default units of vector acceleration and vector deceleration are encoder counts or steps per second per second. The default units of velocity override is a percentage the setting for vector velocity.

```
// Motion settings (GetDlgItemDouble() is a helper function defined
// elsewhere)
//
case IDOK:
    MCGetContourConfig( hCtrlr, iAxis, &Motion );
    Contour.Vector.Accel    = GetDlgItemDouble( hDlg, IDC_TXT_ACCEL );
    Contour.Vector.Decel    = GetDlgItemDouble( hDlg, IDC_TXT_DECEL );
    Contour.Vector.Velocity = GetDlgItemDouble( hDlg, IDC_TXT_VELOCITY );
    Contour.VelocityOverride = GetDlgItemDouble( hDlg, IDC_TXT_MAX_TORQUE );
    MCSetContourConfig( hCtrlr, iAxis, &Motion );
```

### Define the type of contour move

The *nMode* parameter of the **MCBlockBegin()** function is used to define the type of contour move to be executed. The following types of contour motion are supported:

<i>nMode</i> parameter	Contour move type	Description
MC_BLOCK_CONTR_USER	User defined, 1 to 6 axes	Specifies that this block is a user defined contour path motion. <i>INum</i> should be set to the controlling axis number.
MC_BLOCK_CONTR_LIN	Linear interpolated move, 1 to 6 axes	Specifies that this block is a linear contour path motion. <i>INum</i> should be set to the controlling axis number.
MC_BLOCK_CONTR_CW	Clockwise arc, 2 axes	Specifies that this block is a clockwise arc contour path motion. <i>INum</i> should be set to the controlling axis number.
MC_BLOCK_CONTR_CCW	Counter Clockwise arc, 2 axes	Specifies that this block is a counter-clockwise arc contour path motion. <i>INum</i> should be set to the controlling axis number.

Examples of a linear move and a clockwise arc follow:

```
// Linear move
//
MCBlockBegin( hCtrlr, MC_BLOCK_CONTR_LIN, 1 );
    MCMoveAbsolute( hCtrlr, 1, 10000.0 );
    MCMoveAbsolute( hCtrlr, 2, 20000.0 );
    MCMoveRelative( hCtrlr, 3, -5000.0 );
MCBlockEnd( hCtrlr, NULL );

// Clockwise arc move
//
MCBlockBegin( hCtrlr, MC_BLOCK_CONTR_CW, 1 );
    MCArcCenter( hCtrlr, 1, MC_CENTER_ABS, 20000.0 );
    MCArcCenter( hCtrlr, 2, MC_CENTER_ABS, 0.0 );
    MCMoveAbsolute( hCtrlr, 1, 40000.0 );
    MCMoveAbsolute( hCtrlr, 2, 0.0 );
MCBlockEnd( hCtrlr, NULL );
```

### Loading the Contour Buffer for Continuous Path Contouring

The DCX Contour Buffer is used to support Continuous Path Contouring. When a single contour move is executed, the axes will decelerate (at the specified vector velocity) and stop at the target. If multiple contour move commands are issued, the contour buffer allows moves to smoothly transition from one to the other. The vector motion will not decelerate and stop until the contour buffer is empty or an error condition (max following error exceeded, limit sensor 'trip', etc...) occurs.

When axis 1 is the controlling axis, up to 256 linear or 128 arc motions (an arc move takes up twice as much buffer space) can be queued up in the Contouring Buffer. If one of the other five axes is the controlling axis, only 16 motions can be queued up. The **MCGetContouringCount()** command will report how many contour moves have been executed since the axes were last configured for contour motion with **MCSetOperatingMode()**. The contouring count is stored as a 32 bit value, which means that 2,147,483,647 contour moves can be executed before the contour count will 'roll over'.

To delay starting contour motion until the contour buffer has been loaded use the **MCEnableSynch()** command. This command should be issued to the controlling axis **before** issuing any contour moves. Moves issued after the **MCEnableSynch()** command will be queued into the contour buffer. To begin executing the moves in the buffer, issue the **MCGoEx()** command to the controlling axis. To return to normal operation (immediate execution of contour move commands), issue **MCEnableSynch()** to the controlling axis with the state = FALSE.

### Multi Axis Linear Interpolated moves

An example of three linear interpolated moves is shown below. Once the first compound move command is issued, motion of the three axes will start immediately (at the specified vector velocity). The other two compound commands are queued into the contouring buffer. As long as additional contour moves reside in the contour buffer continuous path contour motion will occur. In this example, smooth vector motion will continue (without stopping) until all three linear moves have been completed (the contour buffer has been emptied). At this time the axes will simultaneously decelerate and stop.

```
MCSetOperatingMode( hCtrlr, 1, 1, MC_MODE_CONTOUR );
MCSetOperatingMode( hCtrlr, 2, 1, MC_MODE_CONTOUR );
MCSetOperatingMode( hCtrlr, 3, 1, MC_MODE_CONTOUR );
```

---

```
// Motion settings (GetDlgItemDouble() is a helper function defined
// elsewhere)
//
case IDOK:
    MCGetContourConfig( hCtrlr, iAxis, &Motion );
    Contour.Vector.Accel      = GetDlgItemDouble( hDlg, IDC_TXT_ACCEL );
    Contour.Vector.Decel      = GetDlgItemDouble( hDlg, IDC_TXT_DECEL );
    Contour.Vector.Velocity    = GetDlgItemDouble( hDlg, IDC_TXT_VELOCITY );
    Contour.VelocityOverride   = GetDlgItemDouble( hDlg, IDC_TXT_MAX_TORQUE );
    MCSetContourConfig( hCtrlr, iAxis, &Motion );

// Linear move #1
//
MCBlockBegin( hCtrlr, MC_BLOCK_CONTR_LIN, 1 );
    MCMoveAbsolute( hCtrlr, 1, 85000.0 );
    MCMoveRelative( hCtrlr, 2, 12000.0 );
    MCMoveAbsolute( hCtrlr, 3, -33000.0 );
MCBlockEnd( hCtrlr, NULL );

// Linear move #2
//
MCBlockBegin( hCtrlr, MC_BLOCK_CONTR_LIN, 1 );
    MCMoveAbsolute( hCtrlr, 1, 0.0 );
    MCMoveAbsolute( hCtrlr, 2, 0.0 );
    MCMoveAbsolute( hCtrlr, 3, 0.0 );
MCBlockEnd( hCtrlr, NULL );

// Linear move #3
//
MCBlockBegin( hCtrlr, MC_BLOCK_CONTR_LIN, 1 );
    MCMoveAbsolute( hCtrlr, 1, 5000.0 );
    MCMoveRelative( hCtrlr, 2, 23000.0 );
    MCMoveAbsolute( hCtrlr, 3, -16000.0 );
MCBlockEnd( hCtrlr, NULL );
```

## Arc Motion

The DCX supports specifying an arc motion in two axes in any of three different ways:

- Specify center and end point
- Specify radius and end point (not supported by MCAPI)
- Specify center and ending angle (not supported by MCAPI)

When the first arc motion is issued, motion of the two axes will start immediately (at the specified vector velocity). Additional contour motions will be queued into the contouring buffer. As long as additional contour moves reside in the contour buffer continuous path contour motion will occur. In this example, smooth vector motion will continue (without stopping) until all both arc motions have been completed (the contour buffer has been emptied). At this time the axes will simultaneously decelerate and stop.

### Arc motions by specifying the center point and end point

The **MCArcEnter()** command is used to specify the center position of the arc. This command also defines which two axes will perform the arc motion. The **MCMoveAbsolute()** or **MCMoveRelative()** commands are used to specify the end point of the arc. A spiral motion will be performed if the

distance from the starting point to center point is different than the distance from the center point to end point. An example of two arc motions is shown below:

```
MCSetOperatingMode( hCtrlr, 1, 1, MC_MODE_CONTOUR );
MCSetOperatingMode( hCtrlr, 2, 1, MC_MODE_CONTOUR );
MCSetOperatingMode( hCtrlr, 3, 1, MC_MODE_CONTOUR );

// Motion settings (GetDlgItemDouble() is a helper function defined
// elsewhere)
//
case IDOK:
    MCGetContourConfig( hCtrlr, iAxis, &Motion );
    Contour.Vector.Accel      = GetDlgItemDouble( hDlg, IDC_TXT_ACCEL );
    Contour.Vector.Decel      = GetDlgItemDouble( hDlg, IDC_TXT_DECEL );
    Contour.Vector.Velocity    = GetDlgItemDouble( hDlg, IDC_TXT_VELOCITY );
    Contour.VelocityOverride   = GetDlgItemDouble( hDlg, IDC_TXT_MAX_TORQUE );
    MCSetContourConfig( hCtrlr, iAxis, &Motion );

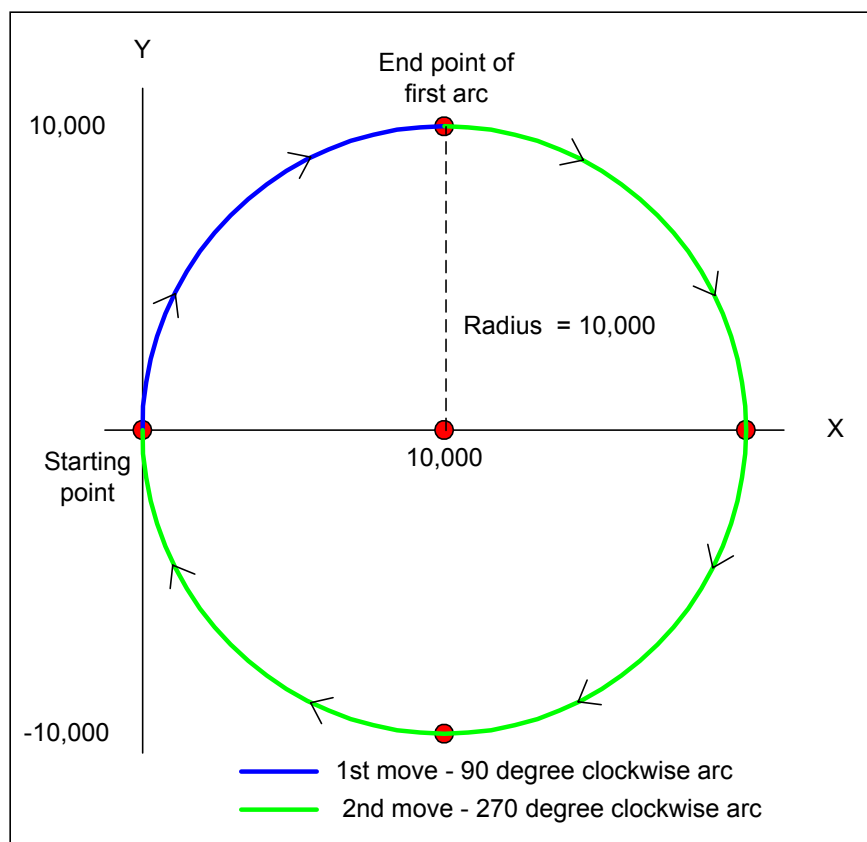
// Clockwise arc move #1
//
MCBlockBegin( hCtrlr, MC_BLOCK_CONTR_CW, 1 );
    MCArcCenter( hCtrlr, 1, MC_CENTER_ABS, 10000.0 );
    MCArcCenter( hCtrlr, 2, MC_CENTER_ABS, 0.0 );
    MCMoveAbsolute( hCtrlr, 1, 20000.0 );
    MCMoveAbsolute( hCtrlr, 2, 0.0 );
MCBlockEnd( hCtrlr, NULL );

// Clockwise arc move #2
//
MCBlockBegin( hCtrlr, MC_BLOCK_CONTR_CCW, 1 );
    MCArcCenter( hCtrlr, 1, MC_CENTER_REL, -10000.0 );
    MCArcCenter( hCtrlr, 2, MC_CENTER_REL, 0.0 );
    MCMoveRelative( hCtrlr, 1, -20000.0 );
    MCMoveRelative( hCtrlr, 2, 0.0 );
MCBlockEnd( hCtrlr, NULL );
```

### Arc motions by specifying the radius and end point

The ***define the Radius of an arc*** (aRRn) command is used to execute an arc move by specifying the radius of an arc. The axis specifier *a* should be the controlling axis for the contour move. The parameter *n* should equal the radius of the arc. If the arc is greater than 180 degrees, the parameter must be expressed as a negative number.

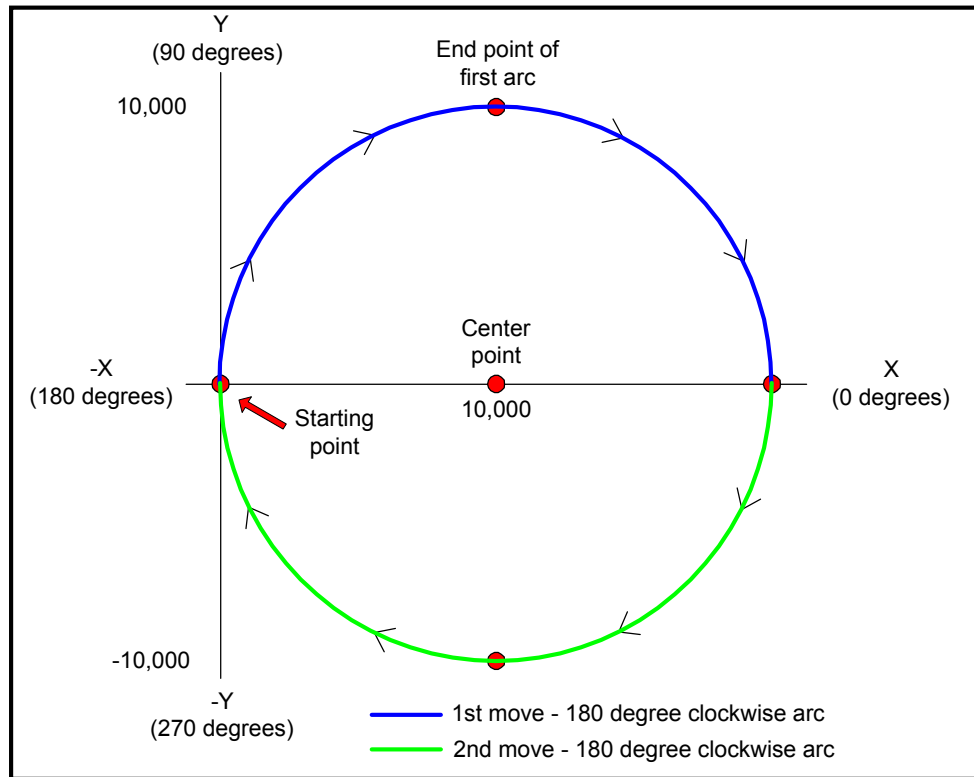
```
1CM1,2CM1                                ;define axis 1 as controlling axis
1CP2,1MR10000,2MR10000,1RR10000          ;90° clockwise arc, radius = 10000
1CP2,1MR-10000,2MR-10000,1RR-10000       ;270° degree arc, radius = 10000,
                                           ;negative radius parameter indicates
                                           ;arc greater than 180°
```



### Arc motions by specifying the center point and ending angle

The **define the Ending angle (absolute) of an arc** (aEAn) and **define the Ending angle (relative) of an arc** (aERn) commands can be used in conjunction with the **define the Center (absolute) of an arc** (CA) and **define the Center (relative) of an arc** (CR) commands to execute an arc motion. When using this method to specify an arc, the move absolute and move relative commands are not used. The center point commands define the radius of the arc. The ending arc angle commands define the end point of the arc as an angle relative to the X axis.

```
1CP2,1CA10000,2CA0,1EA0           ;Clockwise arc motion in X & Y
1CP2,1CR-10000,2CR0,1ER180        ;Clockwise arc motion in X & Y
```



### Changing the velocity 'on the fly'

'On the fly' velocity changes during contour mode motion are accomplished by using the **VelocityOverride** member of the **MCContour** data structure. Issue the command (to the controlling axis) to scale the vector velocity of a linear or arc motion. The rate of change is defined by the current settings for vector acceleration and vector deceleration.



Changing the velocity of a contour group using Velocity Override is not supported for S-curve and/or Parabolic velocity profiles.

### Cubic Spline Interpolation of linear moves

To have the DCX perform 'curve fitting' (cubic spline interpolation) on a series of linear moves, issue the **MCEnableSynch()** command to the controlling axis. Next issue linear contour path commands to points on the curve. After loading the desired number of into the contour buffer, issue a **MCGOEx()** command with the value *Param* set to 1. Motion will continue from the first to the last point in the contour buffer. To return to normal operation, issue the **MCEnableSynch()** command with parameter *pState* = FALSE.



Note that when performing cubic spline interpolation, only **128 motions** can be queued up if **axis 1 is the controlling axis**. If the controlling axis is not axis one, **only 16 motions** can be queued up in the controller.



### User Defined Contour path

For applications where orthogonal geometry is not applicable, the DCX allows the user to define a custom contour distance. In a typical X, Y, and Z axis linear move, the DCX controller will calculate the total length of the contour distance as follows:

Beginning position: X=0, Y=0, Z=0

Target position: X=10,000, Y=10,000, Z=1000

$$\begin{aligned}
 \text{Calculated Contour distance} &= \sqrt{X^2 + Y^2 + Z^2} \\
 &= \sqrt{(10,000^2 + 10,000^2 + 5^2)} \\
 &= \sqrt{(100,000,000 + 100,000,000 + 1,000,000)} \\
 &= \sqrt{201,000,000} \\
 &= 14177.44
 \end{aligned}$$

The DCX would then use the settings for vector velocity, vector acceleration, and vector deceleration to calculate the trajectory of the move. When a User Defined Contour Path is selected (**MCBlockBegin** with parameter *nMode* set to **MC\_BLOCK\_CONTR\_USER**), the **MCContourDistance( )** command is used to enter the non-orthogonal contour distance.

### Special case: setting the Maximum Velocity of an Axis

When executing simple point to point or velocity mode motions the maximum velocity of each axis is set individually. When executing multi axis contour moves, the maximum velocity is typically expressed as the velocity of the 'end effector' of the contour group. In a cutting application the 'end effector' would be the tool doing the cutting (torch, laser, knife, etc...). Setting the maximum velocity of an axis in the contoured group is not typically supported.

By combining a user define contour path (**MCBlockBegin** with parameter *nMode* set to **MC\_BLOCK\_CONTR\_USER**) with the **MCContourDistance( )** command with parameter *Distance* = 0, the user can execute multi axis contour moves and limit the maximum velocity of an individual axis. In this mode of operation the **MCVectorVelocity( )** command is **not** used to set the velocity of the contour group. The axis with the longest move time will define the total time for the contour move. For moves of sufficient distance where the axis has enough time to fully accelerate, this one axis will move at its preset maximum velocity. All other axes will move at velocities lower than their specified maximum. The velocity profiles of the other axes in the contour group will be set such that all axes start and stop at the same time. In the following example, axes one and two are commanded to move the same distance but the maximum velocity for axis two is 1/3 that of axis one. Since both axes are moving the same distance, they will also travel at matching velocities.

```

MCSetVelocity( hCtrlr, 1, 300.0 );
MCSetAcceleration( hCtrlr, 1, 1000.0 );
MCSetDeceleration( hCtrlr, 1, 1000.0 );

MCSetVelocity( hCtrlr, 2, 100.0 );
MCSetAcceleration( hCtrlr, 2, 1000.0 );
MCSetDeceleration( hCtrlr, 2, 1000.0 );

MCSetOperatingMode( hCtrlr, 1, 1, MC_MODE_CONTOUR );
MCSetOperatingMode( hCtrlr, 2, 1, MC_MODE_CONTOUR );

MCContourdistance( hCtrlr, 1, 0.0 );

```

```
MCBlockBegin( hCtrlr, MC_BLOCK_CONTR_USER, 1 );
  MCMoveAbsolute( hCtrlr, 1, 1000.0 );
  MCMoveRelative( hCtrlr, 2, 1000.0 );
MCBlockEnd( hCtrlr, NULL );
```

If the commanded move distance of axis one was 2000 counts it would move at a higher velocity than axis two, but it would not reach its maximum velocity as set by the **MCSetVelocity( )** command.

## Electronic Gearing

The DCX supports slaving any axis or axes to a master. Moving the master axis will cause the slave to move based on the specified slave ratio. The optimal position of the slave axis is calculated by multiplying the optimal position of the master by the gearing ratio of the slave. The slave's optimal position is maintained proportional to the master's position. This can be used in applications where multiple motors drive the same load. Gearing supports both servo and stepper axes, with the master axis operating in jogging, position, velocity or contouring mode. If a following error or limit error occurs on any of the geared axes (master or slaves) all axes in the geared group will stop.



Electronic gearing **does not support S-curve or Parabolic** velocity profiles

The MCAPI function **MCEnableGearing( )** configures and initiates gearing. The DCX supports ratios (*Ratio*) ranging from -65535 to +65536. If the slave ratio is a positive value, a move in the positive direction of the master will cause a move in the positive direction of the slave. If the slave ratio is a negative value, a move in the positive direction of the master will cause a move in the negative direction of the slave. The following program example configures axes 2, 3, and 4 as slaves of axis 1.

```
// Enable gearing of axis 2, 3, and 4
// Move axis 1 (master), slaves (axes 2, 3, and 4) will move at define ratio
MCEnableGearing( hCtrlr, 2, 1, 0.5, TRUE );
MCEnableGearing( hCtrlr, 3, 1, 12.87, TRUE );
MCEnableGearing( hCtrlr, 4, 1, -125, TRUE );
MCMoveRelative( hCtrlr, 1, 215.0 );

// disable gearing
MCEnableGearing( hCtrlr, 2, 1, 0.5, FALSE );
MCEnableGearing( hCtrlr, 3, 1, 12.87, FALSE );
MCEnableGearing( hCtrlr, 4, 1, -125, FALSE );
```



Note – if the slave axes are servo's, the **PID parameters** for each axis **must be defined prior** to beginning master/slave operation.

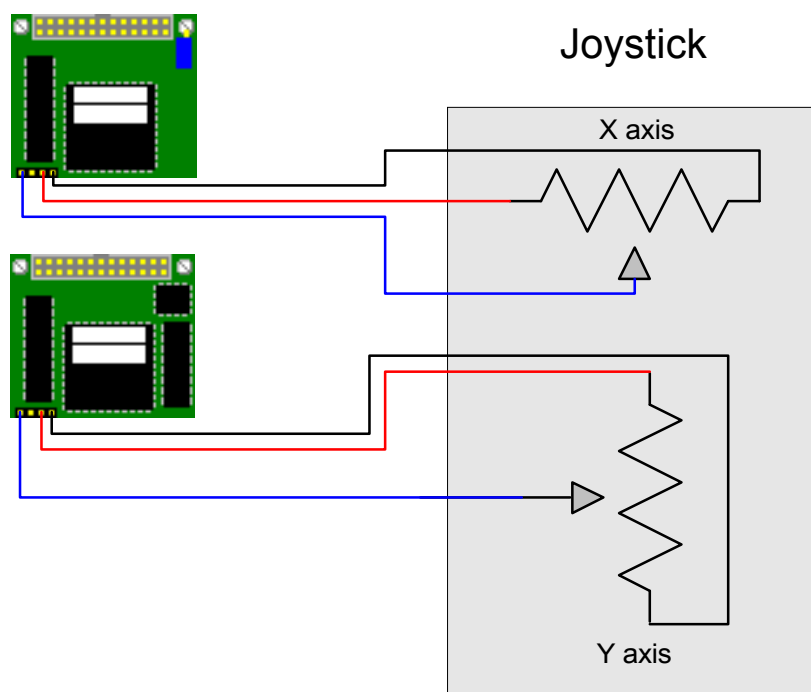


Note – Changing the slave ratio 'on the fly' may cause the mechanical system to 'jerk' or the DCX to 'error out' (following error).

## 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 and steppers with precision at both low and high speeds, all that is required is a manual command input. A joystick provides such an input with natural hand to motion coordination. It can be used for jogging both servo (DCX-MC200, DCX-MC210) and stepper (DCX-MC260) motors on the DCX. The rest of this section describes how to implement joystick control.

Typical joysticks have 2 potentiometers, one for each axis of motion. Each 'pot' of a joystick can be connected to one servo or stepper module. Pots with a total resistance between 10K and 100K ohms are suitable for use with the DCX motor modules. By connecting the end taps of one pot to pins 3 (+5V supply) and 4 (Ground) of a motor module's J4 connector, and the center tap to pin 1, the module is able to read the pot's position.



To test the connection of the joystick to a module's analog input, from WinControl use the Read Float and Tell Register commands as in the following example:

```
1RV200,TR                                ;load the A/D reading of axis #1 into
                                           ;the accumulator, report the A/D value
```

The value that is reported by the Tell Register command is the voltage level at the module's analog input in units of volts. With the joystick at its center position, the reading should be 2.5 VDC. Most joysticks have centering adjustments to correct this reading if necessary. As the joystick is moved from one extreme to the other, the analog input reading should range from 0 to +5.0.

Jogging on the DCX is implemented using the MCAPI **Data Structure MCJOG**. For additional information on the **MCAPI** and **Data Structures** please refer to the **MCAPI on-line help**.

When jogging is enabled on an axis of the DCX with the function **MCEnableJog()**, the readings from the analog input, multiplied by the **Jog.Gain** setting, will cause the servo or stepper to move. Given the desired maximum velocity when the joystick is pushed to the extreme, the appropriate Jog Gain can be calculated by the following equation:

$$\text{Jog Gain} = \text{Desired Velocity} / 2.5$$

For example, if the desired maximum velocity is 1000 counts per second, the Jog Gain should be set to  $1000 / 2.5 = 400$ .

During jogging, the servo or stepper will accelerate and decelerate at the rate specified using the data structure member **Jog.Acceleration**. For closed loop servos, this value is typically set high to provide quick response to the joystick. For stepper motors, the acceleration must be set to a value low enough that the motor will not stall during acceleration. For stepper motors, the **Jog.MinVelocity** data structure member will enhance the response of the motor to the joystick. This command will have no effect on servo motors.

Jogging of the axes is enabled using the **MCEnableJog** data structure member. After calling this function the axes will move with a velocity proportional to the amount the joystick is deviated from its center position. To increase or decrease the maximum velocity, the Jog Gain can be adjusted up or down at any time. Depressing the joystick in the opposite direction should cause the servo or stepper to reverse direction. To change the direction that a servo or stepper moves in response to the joystick, set the Jog Gain to a negative value.

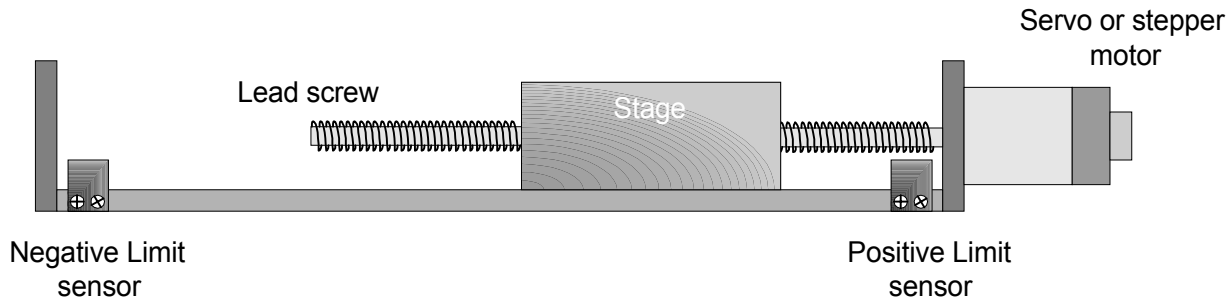
Because of mechanical and electrical variations, the joystick reading may 'bounce'. This will cause undesirable drifting of the axes when the joystick is enabled. The **Jog.Deadband** data structure member is used to define a voltage range that will not cause the axis to move. Any 'signal bounce' that falls within the range of the jog deadband setting will be ignored and no motion will occur. Any voltage level that exceeds the dead band range, whether due to a change of position of the joystick or signal noise, will cause the axes to move.

Most joysticks have adjustments for setting the center position/voltage. The DCX also has a parameter for adjusting the center position of the joystick. The **Jog.Offset** data structure member can be used to redefine the null (no motion) voltage level of a joystick. This value defaults to 2.5 volts. Changing the Jog Offset to a different value will provide an increased velocity range in one direction, but a reduced range in the opposite direction.

```
MCJOG Jog;                                // declare a JOG data structure
Jog.Gain = 400;                            // define the jog gain
Jog.Acceleration = 10000;                 // define the jog acceleration
Jog.MinVelocity = 150;                    // define the jog minimum velocity
Jog.Deadband = 0.20;                      // define a voltage dead band
Jog.Offset = 2.01JF;                     // define 'null' position as 2.0V
MCSetJogConfig(hCtrlr, 1, &JOG);        // set jog parameters
MCEnableJog(hCtrlr, 1, TRUE);            // start jogging
```

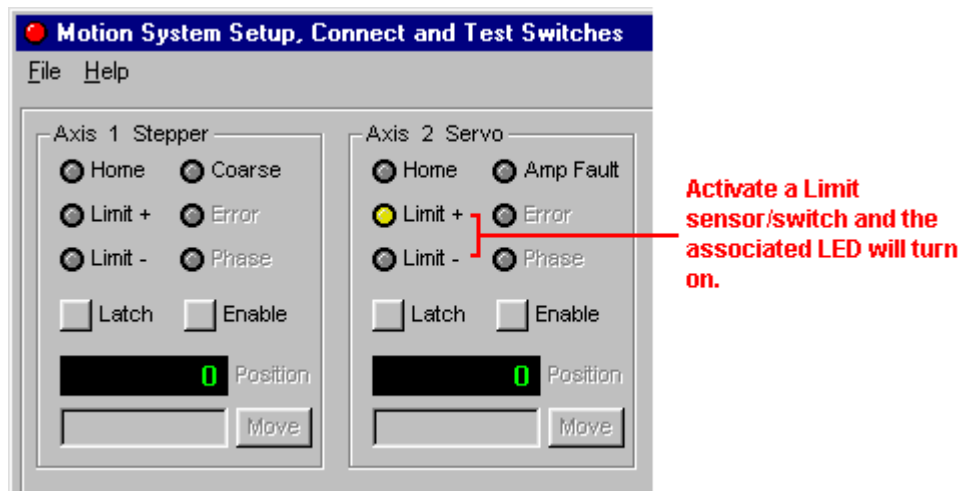
## Defining Motion Limits

The DCX Motion Controller implements two types of motion limits error checking. End of travel or 'Hard' limit switch/sensor inputs and 'soft' user programmable position limits.



### Hard Limits

The Limit + /- inputs of all MC2XX 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 MC2XX 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( hCtrlr, 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( hCtrlr, 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:

<b>Flag</b>	<b>Description</b>
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( hCtrlr, 3, MC_LIMIT_BOTH | MC_LIMIT_ABRUPT, MC_LIMIT_BOTH |
MC_LIMIT_SMOOTH, 10.0, 490.0 );
```

## Homing Axes

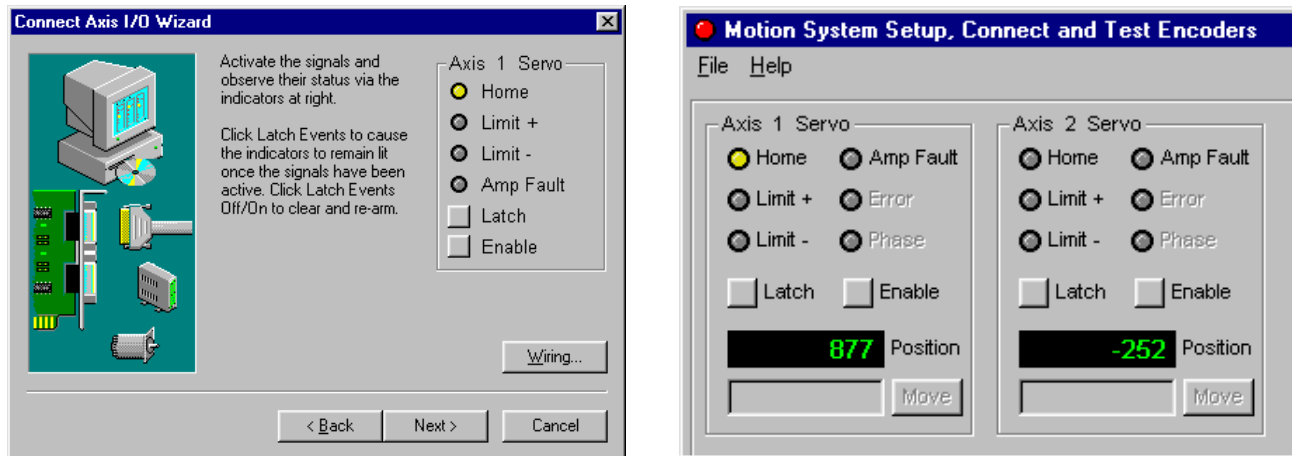
When power is applied or the DCX is reset, the current position of all servo and stepper axes are initialized to zero. If they are subsequently moved, the controller will report their positions relative to the position where they were last initialized. At any time the user can call the **MCSetPosition(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

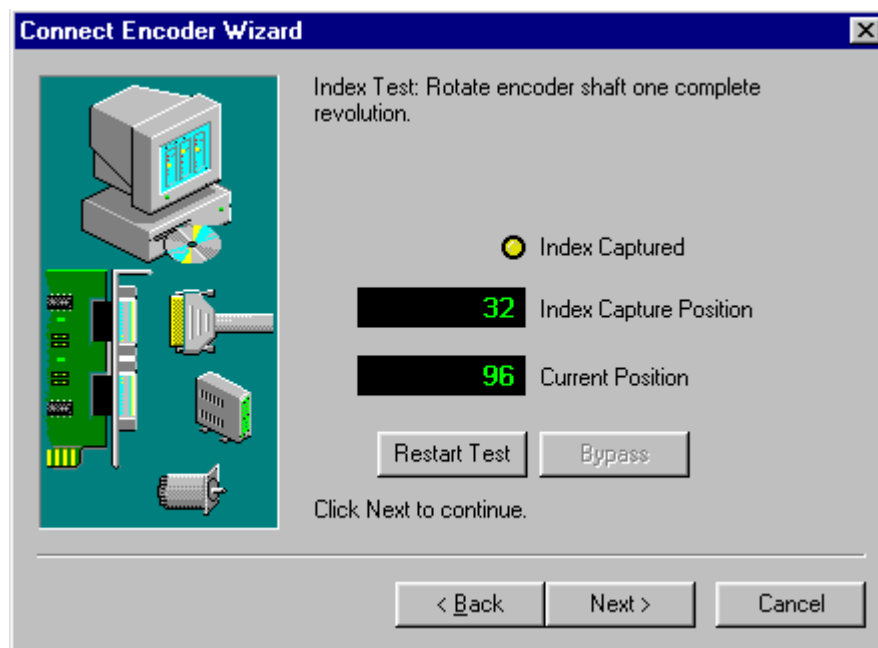
### Verifying the operation of the Home Sensor

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( hCtrlr, 1, 0, MC_MODE_VELOCITY );
MCDirection( hCtrlr, 1, MC_DIR_POSITIVE );
MCSetVelocity( hCtrlr, 1, 5000.0 );
MCGo( hCtrlr, 3 );

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

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

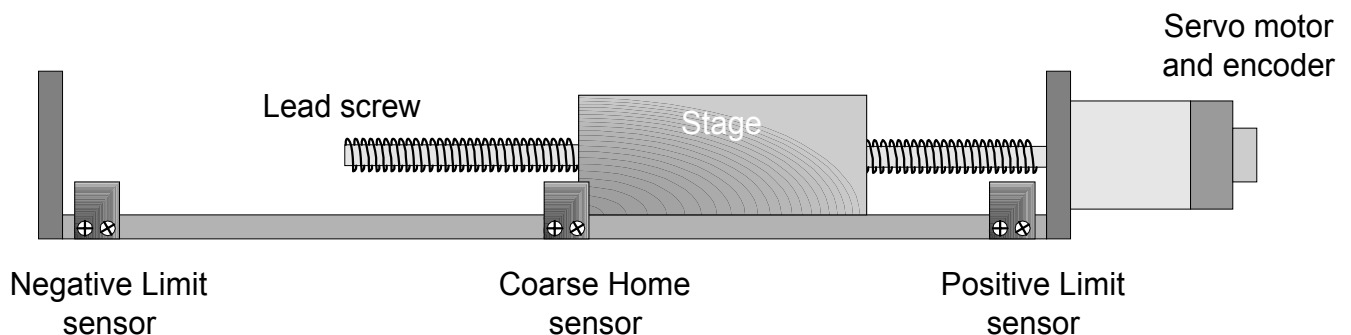
```

### 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( hCtrlr, 1, MC_LIMIT_SMOOTH | MC_LIMIT_HIGH | MC_LIMIT_LOW, 0, 0, 0
);
MCSetOperatingMode( hCtrlr, 1, 0, MC_MODE_VELOCITY );
MCSetVelocity( hCtrlr, 1, 10000.0 );
MCDirection( hCtrlr, 1, MC_DIR_POSITIVE );
MCGoEx( hCtrlr, 1, 0.0 );

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

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

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

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

// Issue position mode move to location of index mark (position 0)
//
MCSetOperatingMode( hCtrlr, 1, 0, MC_MODE_POSITION );
MCEnableAxis( hCtrlr, 1, TRUE );
MCMoveAbsolute( hCtrlr, 1, 0.0 );
MCWaitForStop( hCtrlr, 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,1RL0,IS25,MJ11,NO,IS17,MJ12,NO,JR-7
                                   ;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
MD12,1MN,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( hCtrlr, 1, MC_LIMIT_SMOOTH | MC_LIMIT_HIGH | MC_LIMIT_LOW, 0, 0, 0
);
MCSetOperatingMode( hCtrlr, 1, 0, MC_MODE_VELOCITY );
MCSetVelocity( hCtrlr, 1, 10000.0 );
MCDirection( hCtrlr, 1, MC_DIR_POSITIVE );
MCGoEx( hCtrlr, 1, 0.0 );

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

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

// Stop the axis and define the leading edge of the limit switch as position 0
//
MCAbort( hCtrlr, 1 );

```

```

MCWaitForStop( hCtrlr, 1, 0.1 );
MCSetPosition( hCtrlr, 1, 0.0 );
MCSetOperatingMode( hCtrlr, 1, 0, MC_MODE_POSITION );
MCEnableAxis( hCtrlr, 1, TRUE );
MCMoveAbsolute( hCtrlr, 1, -100.0 );

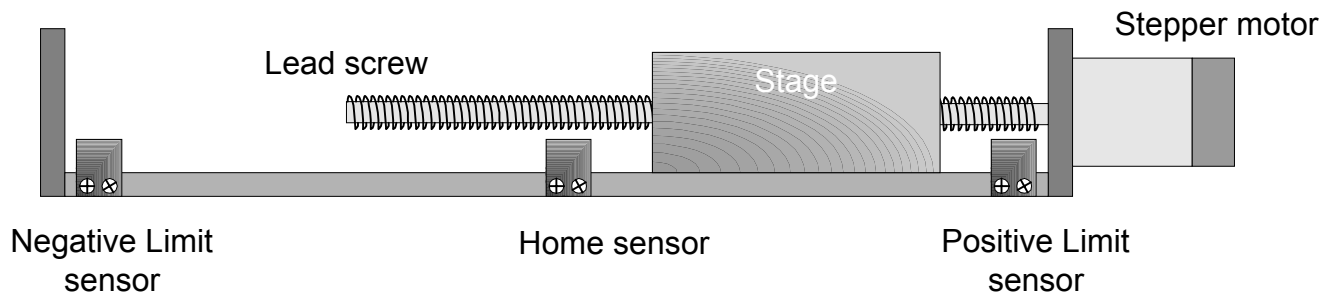
; MCCL homing sequence (using positive limit, coarse home, and index mark)

MD1,1LM2,1LN3,MJ10 ;call homing macro
MD10,1VM,1DI0,1GO,1RL0,IS17,MJ11,NO,JR-4
;move and poll the Limit + sensor
MD11,1MN,1DI1,1SV1000,1GO,1RL0,IC17,MJ12,NO,RP-4
;move negative until limit + inactive
MD13,1AB,1WS.1,1DH0,1PM,1MN,1MA-100
;stop when limit + not active, define
;position as 0. Move to position -100.

```

### Homing open loop steppers

Open loop steppers are typically homed based on the position of a home sensor. Unlike servos that use a precision reference index mark, steppers are more prone to homing inaccuracies due to the lower repeatability of the sensor output. To achieve the highest possible repeatability; reduce the velocity of the axis and always approach the home sensor from the same direction. Here is a typical linear axis controlled by a stepper motor. A home sensor defines the home position of the axis. End of travel or Limit Switches are used to protect against damage of the mechanical components.



When power is applied or the DCX is reset, the position of the stage is unknown. The following command sequence will move the stage in the positive direction. If the home sensor 'goes active' before the positive limit sensor, the Find Edge command will redefine the reported position of the axis. If the positive limit sensor 'goes active', the stage will change direction, until home sensor is located. The Find Edge command is then used to redefine the position of the axis.

```

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

//
// Wait for home or positive limit inputs

```

---

```

dwStatus = MCGetStatus( hCtrlr, 1);
while (! MCDecodeStatus( hCtrlr, dwStatus, MC_STAT_INP_HOME) ||
       ! MCDecodeStatus( hCtrlr, dwStatus, MC_STAT_PLIM_TRIP)) {
    dwStatus = MCGetStatus( hCtrlr, 1);
}

// If positive limit switch active
//
dwStatus = MCGetStatus( hCtrlr, 1);
if (!MCDecodeStatus( hCtrlr, dwStatus, MC_STAT_PLIM_TRIP)) {
    MCEnableAxis( hCtrlr, 1, TRUE );
    MCSetVelocity( hCtrlr, 1, 10000.0 );
    MCDirection( hCtrlr, 1, MC_DIR_NEGATIVE );
    MCGoEx( hCtrlr, 1, 0.0 );
    MCFindEdge( hCtrlr, 1, 0.0 );
    MCStop( hCtrlr, 1 );
    MCWaitForStop( hCtrlr, 1, 0.1 );
    MCEnableAxis( hCtrlr, 1, FALSE );
    MCWait( hCtrlr, 0.1 );
    MCEnableAxis( hCtrlr, 1, TRUE );
    MCDirection( hCtrlr, 1, MC_DIR_POSITIVE );
    MCSetVelocity( hCtrlr, 1, 5000.0 );
    MCGoEx( hCtrlr, 1, 0.0 );
    while (! MCDecodeStatus( hCtrlr, dwStatus, MC_STAT_INP_HOME)) {
        dwStatus = MCGetStatus( hCtrlr, 1);
    }
    MCStop( hCtrlr, 1 );
    MCWaitForStop( hCtrlr, 1, 0.1 );
}

// If Home sensor active
//
dwStatus = MCGetStatus( hCtrlr, 1);
if (!MCDecodeStatus( hCtrlr, dwStatus, MC_STAT_INP_HOME)) {
    MCStop( hCtrlr, 1 );
    MCWaitForStop( hCtrlr, 1, 0.1 )
}
MCDirection( hCtrlr, 1, MC_DIR_POSITIVE );
MCSetVelocity( hCtrlr, 1, 5000.0 );
MCGoEx( hCtrlr, 1, 0.0 );
while (! MCDecodeStatus( hCtrlr, dwStatus, MC_STAT_INP_HOME)) {
    dwStatus = MCGetStatus( hCtrlr, 1);
}
MCStop( hCtrlr, 1 );
MCWaitForStop( hCtrlr, 1, 0.1 );

// Find the leading edge of the Home sensor and define position 0
//
MCDirection( hCtrlr, 1, MC_DIR_NEGATIVE );
MCSetVelocity( hCtrlr, 1, 1000.0 );
MCGoEx( hCtrlr, 1, 0.0 );
MCFindEdge( hCtrlr, 1, 0.0 );
MCStop( hCtrlr, 1 );
MCWaitForStop( hCtrlr, 1, 0.1 )

// Enable axis to re-initialize the position register. Issue position mode
move to location of index mark (position 0)
//
MCSetOperatingMode( hCtrlr, 1, 0, MC_MODE_POSITION );
MCEnableAxis( hCtrlr, 1, TRUE );
MCMoveAbsolute( hCtrlr, 1, 0.0 );
MCWaitForStop( hCtrlr, 1, 0.1 );

```

---

```
// MCCL homing sequence (using the home and positive limit switches)

MD1,1LM2,1LN3,MJ10          ;call homing macro
MD10,1VM,1DI0,1SV10000,1GO,1RL0,IS24,MJ11,NO,IS17,MJ13,NO,JR-7
                        ;test for sensors (home and +limit)
MD11,1ST,1WS.1,1DI0,1SV5000,1GO,1RL0,IC24,MJ12,NO,JR-4
                        ;Move positive until home sensor off
MD12,1ST,1WS.1,1DI1,1SV5000,1GO,MJ15
                        ;move back to the home sensor
MD13,1MN,1DI1,1SV5000,1GO,MJ15      ;move out of limit sensor range back
                        ;toward the home sensor
MD14,1FE0,1ST,1WS.1,1MF,WA.1,1MN,1PM,1MA0
                        ;find the active edge of the home                ;sensor. Stop axis,
initialize                        ;position, move to position 0.
```

An axis can be homed even if no home sensor is available. This method of homing utilizes one of the limit (end of travel) sensors to also serve as a home reference. The following command sequences will home an axis at the location where the positive limit sensor 'goes active':

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

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

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

// Stop the axis and define the leading edge of the limit switch as position 0
//
MCAbort( hCtrlr, 1 );
MCWaitForStop( hCtrlr, 1, 0.1 );
MCSetPosition( hCtrlr, 1, 0.0 );
MCSetOperatingMode( hCtrlr, 1, 0, MC_MODE_POSITION );
MCEnableAxis( hCtrlr, 1, TRUE );
MCMoveAbsolute( hCtrlr, 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,1RL0,IS17,MJ11,NO,JR-4
                                ;move and poll the Limit + sensor
MD11,1MN,1DI1,1SV1000,1GO,1RL0,IC17,MJ12,NO,RP-4
                                ;move negative until limit + inactive
MD13,1AB,1WS.1,1DH0,1PM,1MN,1MA-100 ;stop when limit + not active, define
                                ;position as 0. Move to position -100.

```



An axis can be homed even if no index mark or coarse home sensor is available. This method of homing uses 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.

## Homing closed loop steppers

Homing a closed loop stepper is similar to homing an axis with an auxiliary encoder , see the description of **Auxiliary Encoder** in the **Application Solutions** chapter. The Index mark of the encoder is connected to the Auxiliary Encoder Index – input pin (connector J3 pin 22). A Coarse Home sensor is connected to the Auxiliary Encoder Coarse Home input (connector J3 pin 23). This device is used to qualify which index mark pulse is to be used for homing.

```

MCBlockBegin( hCtrlr, MC_BLOCK_COMPOUND, 0 );

// Enable the axis, place it in velocity mode, begin the move. After the Edge
// (Coarse Home Input), reduce the velocity and wait for the index mark to be
// captured. Move to the location of the index mark, set the position of the
// auxiliary encoder
//

MCEnableAxis( hCtrlr, 1, TRUE );
MCSetOperatingMode( hCtrlr, 1, 0, MC_MODE_VELOCITY );
MCSetVelocity( hCtrlr, 1, 1000.0 );
MCDirection( hCtrlr, 1, MC_DIR_POSITIVE );
MCGo( hCtrlr, 1 );
MCWaitForEdge( hCtrlr, TRUE )
MCSetVelocity( hCtrlr, 1, 500.0 );
MCFindAuxEncIndex( hCtrlr, 1, 0.0 );
dwStatus = MCGetStatus( hCtrlr, 1 );
while ( ! MCDecodeStatus( hCtrlr, dwStatus, MC_STAT_INP_AUX ) )
    dwStatus = MCGetStatus( hCtrlr, 1 );
MCGetPositionEx( hCtrlr, 1, &CapturedPosition );
MCStop( hCtrlr, 1 );
MCWaitForStop( hCtrlr, 1, 0.1 );
MCMoveAbsolute( hCtrlr, 1, &CapturedPosition );
MCWaitForStop( hCtrlr, 1, 0.1 );
MCSetAuxEncPos( hCtrlr, 1, 0.0 );
MCBlockEnd( hCtrlr, NULL );

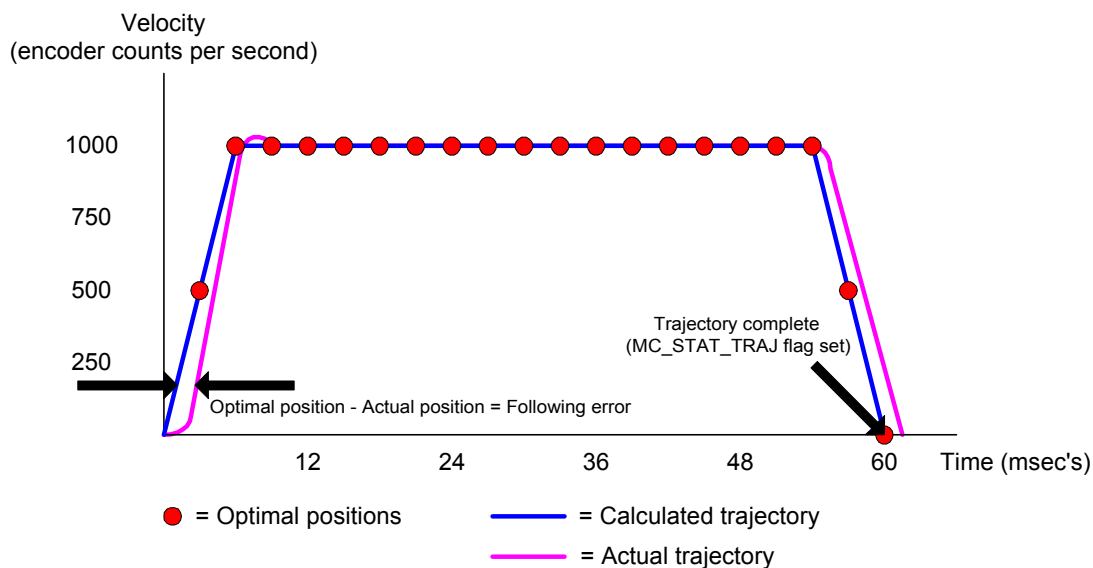
```

## Motion Complete Indicators

When the DCX receives a move command, the Trajectory Generator calculates a velocity profile. This profile is based on:

- The target position (absolute or relative)
- The user defined trajectory parameters (velocity, acceleration, and deceleration)

The velocity profile, as calculated by the DCX trajectory generator, is made up by a series of 'Optimal Positions' that are evenly spaced along the motion path in increments of 4 msec's. These 4 msec optimal positions are passed to the DCX servo modules, which then performs a linear interpolation at the selected servo loop rate.



When 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. 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( hCtrlr, 2, 500.0 );           // move 500 counts
MCWaitForStop( hCtrlr, 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( hCtrlr, 1, 1250.0 );    // move 1250 counts
MCWaitForTarget( hCtrlr, 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
- Acceleration
- Deceleration
- Servo PID parameters

Changes made to any or all of these motion settings while an axis is moving will take affect within 4 msec's.



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



Note – S-curve and Parabolic velocity profiles do not support 'on the fly' changes of target, velocity, acceleration and/or deceleration .



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).



While moving with Parabolic or S-curve profiles, on the fly changes will cause the axis to first decelerate to a stop, then resume motion.

## Feed Forward (Velocity, Acceleration, Deceleration)

Feed forward is a method in which the controller increases the command output to a servo in order to reduce the following error of an axis. Traditionally feed forward is associated with servo systems that use velocity mode amplifiers, but simple current mode amplifiers used for high velocity and high rate of change applications can also benefit from the use of feed forward.

The basic concept of feed forward is to match the servo command voltage output of the controller to a specific velocity of axis. This essentially adds a user defined offset to the digital PID filter, resulting in more accurate motion by reducing the following error. For example:

The maximum velocity of an axis is 500,000 encoder counts per second. With a typical load applied, the user determines that a servo command voltage of 8.25V will cause the motor to rotate at 500,000 encoder counts per second. The feed forward algorithm used by the DCX to generate the servo command output is:

$$\text{DCX output} = \text{Velocity (encoder counts/sec)} \times \text{Feed forward term (encoder counts/volt/sec.)}$$

with a velocity of 500,000 counts per second at a command input of 8.25V the algorithm will be:

$$8.25 \text{ volts} = 500,000 \text{ counts/sec.} \times \text{Feed forward term (encoder counts} \times \text{volt/sec.)}$$

$$\text{Feed forward} = 8.25\text{V} / 500,000 \text{ counts per sec.}$$

$$0.0000165 = 10 \text{ volts} / 100,000 \text{ counts per sec.}$$

```
1VG0.0000165                ;set velocity gain (velocity feed
                             ;forward ) with MCCL command

// set velocity gain (velocity feed forward) using MCAPI function
//
MCGetFilterConfig( hCtrlr, iAxis, &Filter );
Filter.VelocityGain = ( hCtrlr, 1, 0.0000165 );
MCSetFilterConfig( hCtrlr, iAxis, &Filter );
```



An axis that has been tuned without feed forward will need to be **re-tuned** when the feed forward has been changed to a non zero value.

See the description of Tuning a Velocity Mode amplifier in the **Tuning the Servo** section of the **Motion Control** chapter

When feed forward is incorporated into the digital PID filter it becomes the primary component in generating the servo command output voltage. Typically the setting of the other terms of the filter will be:

Proportional gain – reduced by 25% to 50%

Integral gain – reduced by 5% to 25%

Derivative gain – set to zero, if the axis is too responsive reduce the gain of the amplifier

### Acceleration and Deceleration Feed Forward

For most applications, velocity feed forward is sufficient for accurately positioning the axis. However for applications that require a very high rate of change, acceleration and deceleration gain must be used to reduce the following error at the beginning and end of a move.

Acceleration and deceleration feed forward values are calculated using a similar algorithm as used for velocity gain. The one difference is the velocity is expressed as encoder counts per second, while acceleration and deceleration are expressed as encoder counts per second per second.

$$\text{DCX output} = \text{Accel./Decel. (encoder counts/sec/sec.)} * \text{Feed forward term (encoder counts * volt/sec./sec.)}$$



Acceleration and deceleration feed forward values should be set prior to using the Servo Tuning Utility to set the proportional and integral gain.

## 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 and stepper 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**

---

- Auxiliary Encoders
- Backlash Compensation
- Emergency Stop
- Encoder Rollover
- Flash Memory Firmware Upgrade
- Laser Cutting
- Learning/Teaching Points
- Record and Display Motion Data
- Manually Resetting the DCX
- Tangential Knife Control
- Threading Operations
- Torque Mode Output Control
- Defining User Units
- DCX Watchdog

## Application Solutions

---

### Auxiliary Encoders

Servo systems typically use an encoder for position feedback. The encoder is usually mounted to the motor housing and the glass scale of the encoder is coupled directly to the shaft of the motor. This direct coupling provides the DCX with position feedback of the motor shaft, allowing the controller to position the shaft of the motor independent of external mechanical inaccuracies (slipping belts, gear backlash, lead screw runout).

However the ‘task at hand’ of most motion control applications is not to rotate the shaft of a motor, it is to automate a manual operation. To accomplish this, the shaft of the motor is connected to the external mechanics that will actually be doing the work. Take for example a pick and place machine with axes X, Y, and Z. Due to a myriad of gears, pulleys, belts, and lead screws there may be no more than a ‘loose’ association between the motor shaft of the X axis and the actual position of the X axis’ ‘end effector’. This is where an auxiliary encoder can be used to significantly improve the positioning accuracy of a servo or stepper system.

#### Servo Axes with Auxiliary Encoders

An auxiliary encoder is required when the user must reposition an axis to compensate for the discontinuity between the motor shaft and the mechanics that position the ‘end effector’.



While similar in connections, the operation and configuration of a servo and auxiliary encoder is significantly different from a Dual Loop Servo. For a description, please refer to the **Dual Loop Servo** section of the **Motion Control** chapter.

Typically an auxiliary encoder is added to a closed loop servo to allow the user to retrieve the position of the ‘end effector’ **at the end of a move**. The position of the auxiliary encoder is not a component of

the servo command output as calculated by the digital PID filter. The auxiliary encoder is used to determine whether or not the axis is properly positioned.

```
// After a move compare the target and auxiliary encoder position.
// If short of the target, execute a move = the difference of the target &
// encoder position

MCMoveAbsolute( hCtrlr, 1, 1675.5 );
MCWaitForStop( hCtrlr, 1, 0.01 );
if (MCGetTargetEx( hCtrlr, 2, &Target ) == MCERR_NOERROR )
if (MCGetAuxEncPosEx( hCtrlr, 1, &Position ) == MCERR_NOERROR)
    if (Position < 1674.0)
        (Target - Position = AuxEncDiff)
        MCMoveRelative( hCtrlr, 1, AuxEncDiff );
        MCWaitForStop( hCtrlr, 1, 0.01 );
if (MCGetAuxEncPosEx( hCtrlr, 1, &Position ) == MCERR_NOERROR)
    if (Position < 1675.0)
        . . . // print error message
```

### Open Loop Stepper Axes with Auxiliary Encoders

An auxiliary encoder may be used in conjunction with a stepper motor to provide verification of a move. The advantages of an open loop stepper over a closed loop axis are:

- The output pulse train of an open loop system is much more stable
- Easier to configure - open loop systems require no tuning

Typically an encoder is added to an open loop stepper to allow the user to retrieve the encoder position **at the end of a move**. The reported position of the auxiliary encoder is used to determine whether or not the axis is properly positioned.

```
// After a move compare the target and auxiliary encoder position.
// If short of the target, execute a move = the difference of the target &
// encoder position

MCMoveAbsolute( hCtrlr, 1, 122.5 );
MCWaitForStop( hCtrlr, 1, 0.001 );
if (MCGetTargetEx( hCtrlr, 2, &Target ) == MCERR_NOERROR )
if (MCGetAuxEncPosEx( hCtrlr, 1, &Position ) == MCERR_NOERROR)
    if (Position < 122.0)
        (Target - Position = AuxEncDiff)
        MCMoveRelative( hCtrlr, 1, AuxEncDiff );
        MCWaitForStop( hCtrlr, 1, 0.001 );
if (MCGetAuxEncPosEx( hCtrlr, 1, &Position ) == MCERR_NOERROR)
    if (Position < 122.0)
        ... // print error message
```

For additional information about closed loop stepper motion, please refer to the **Closed Loop Steppers and Homing Axes** sections of the **Motion Control** chapter.

### Homing the Auxiliary Encoder

The auxiliary encoder of a servo or stepper may be homed in one of two ways:

- Home the encoder using the Auxiliary Encoder Index input
- Re-define the position of the auxiliary encoder when the primary axis position is initialized

If the encoder includes an index mark output it is recommended that this signal be used to home the reported position of the auxiliary encoder. The repeatability of a system homed using the index mark will be significantly better than that of a system that uses a mechanical switch/electromechanical sensor. The following programming example will initialize the reported position of the auxiliary encoder at the location of the Index mark:

```
MCBlockBegin( hCtrlr, MC_BLOCK_COMPOUND, 0 );

// Enable the axis, place it in velocity mode, begin the move. After the Edge
// (Coarse Home Input), wait for the index mark to be captured. Move to the
// location of the index mark, set the position of the auxiliary encoder
//
MCEnableAxis( hCtrlr, 1, TRUE );
MCSetOperatingMode( hCtrlr, 1, 0, MC_MODE_VELOCITY );
MCSetVelocity( hCtrlr, 1, 1000.0 );
MCDirection( hCtrlr, 1, MC_DIR_POSITIVE );
MCGo( hCtrlr, 1 );
MCWaitForEdge( hCtrlr, TRUE );
MCFindAuxEncIndex( hCtrlr, 1, 0.0 );
dwStatus = MCGetStatus( hCtrlr, 1 );
while ( ! MCDecodeStatus( hCtrlr, dwStatus, MC_STAT_INP_AUX ) )
    dwStatus = MCGetStatus( hCtrlr, 1 );
MCGetPositionEx( hCtrlr, 1, &CapturedPosition );
MCStop( hCtrlr, 1 );
MCWaitForStop( hCtrlr, 1, 0.1 );
MCMoveAbsolute( hCtrlr, 1, &CapturedPosition );
MCWaitForStop( hCtrlr, 1, 0.1 );
MCSetAuxEncPos( hCtrlr, 1, 0.0 );
MCBlockEnd( hCtrlr, NULL );
```



Unlike the **MCFindIndex()** function, which re-defines the position reported by a servos' encoder, **MCSetAuxEncPos()** does not re-define the position of the auxiliary encoder. **MCSetAuxEncPos()** only arms the capture of the encoder index mark, which is then indicated by the status bit **MC\_STAT\_INP\_AUX** being set.

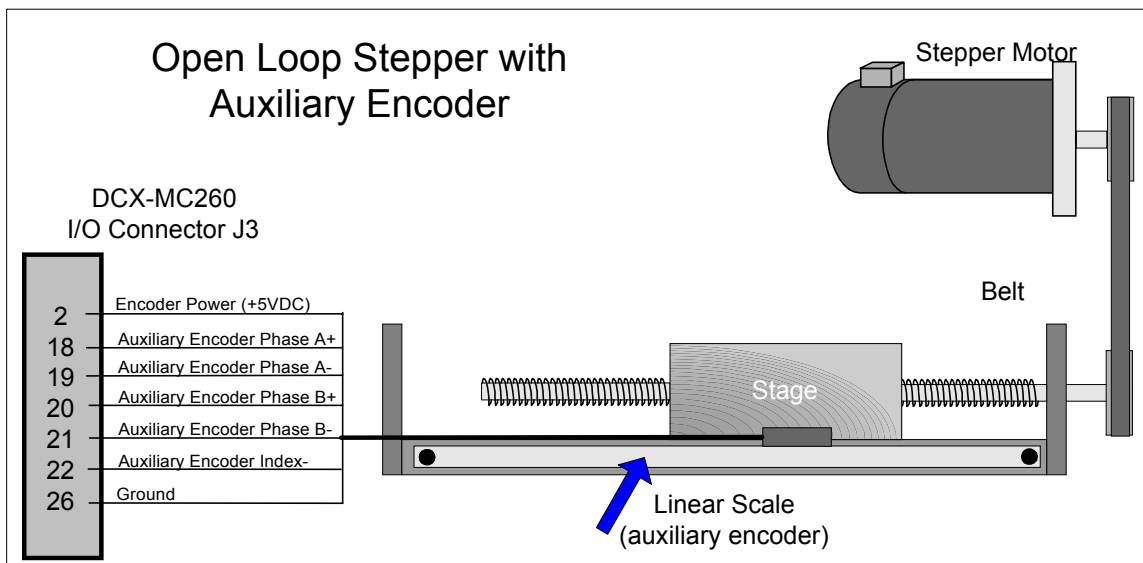
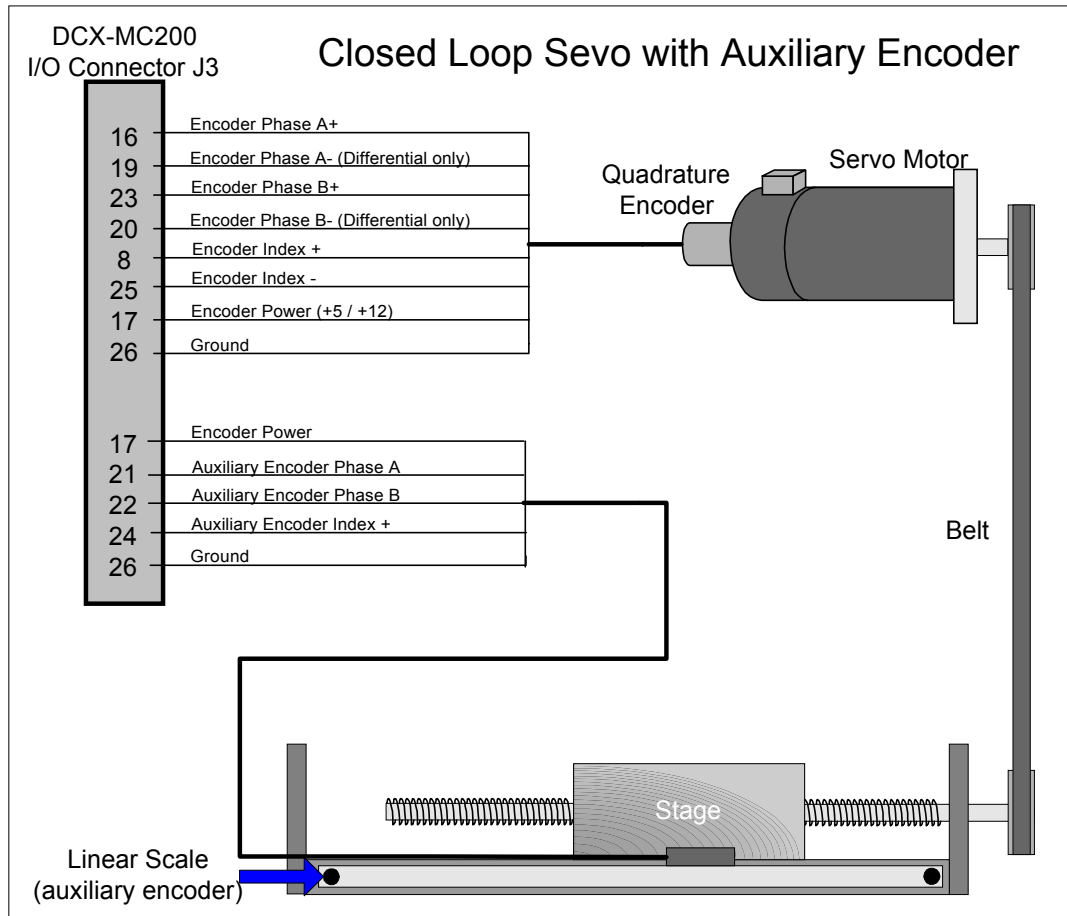
If no encoder index mark output is available, the position of the auxiliary encoder can be redefined at anytime using the MCAPI function **MCSetAuxEncPos()**. The following programming example will re-define the position of the auxiliary encoder of a stepper axis when it is homed.

```
MCBlockBegin( hCtrlr, MC_BLOCK_COMPOUND, 0 );

// Home a stepper axis and re-define the position of the auxiliary encoder
//
MCEnableAxis( hCtrlr, 1, TRUE );
MCSetOperatingMode( hCtrlr, 1, 0, MC_MODE_VELOCITY );
MCSetVelocity( hCtrlr, 1, 1000.0 );
MCDirection( hCtrlr, 1, MC_DIR_POSITIVE );
MCGo( hCtrlr, 1 );
MCFindEdge( hCtrlr, 1, 0.0 );
MCStop( hCtrlr, 1 );
MCWaitForStop( hCtrlr, 1, 0.1 );
MCMoveAbsolute( hCtrlr, 1, 0.0 );
MCWaitForStop( hCtrlr, 1, 0.1 );
MCSetAuxEncPos( hCtrlr, 1, 0.0 );
MCBlockEnd( hCtrlr, NULL );
```

## Auxiliary Encoder Connections

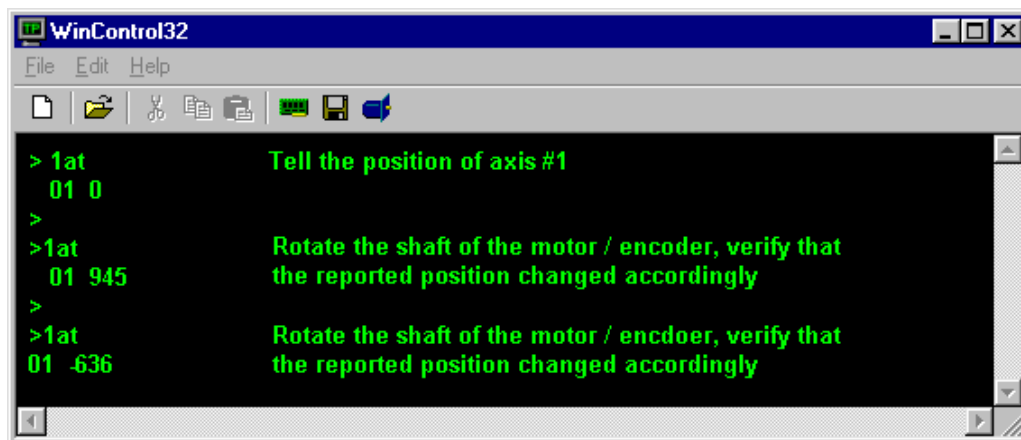
The two diagrams that follow illustrate the typical wiring connections required for interfacing DCX motion control modules to an auxiliary encoder. For additional information please refer to the **Connectors, Jumpers, and Schematics** chapter.





## Verifying the Operation of the Auxiliary Encoder

Use the WinControl MCCL command utility to verify the proper operation of the DCX module and the auxiliary encoder. The example below details testing an encoder connected to axis number one. To test a different axis, issue the **Auxiliary encoder Tell position (aAT)** command with the appropriate prefix (where *a* = axis number).



## Backlash Compensation

In applications where the mechanical system isn't directly connected to the motor, it may be required that the motor move an extra amount to compensate for system backlash. When backlash compensation is enabled, the DCX controller will offset the target position of a move by the user defined backlash distance. This feature is only available for servos (MC200 MC210) at this time.

The function **MCEnableBacklash( )** is used to initiate backlash compensation. The *Backlash* parameter of this function sets the amount of compensation and should be equal to one half of the amount the axis must move to take up the backlash when it changes direction. The units for this command parameter are encoder counts, or the units established by the **MCSetScale( )** command for this axis.

When this feature is enabled, the controller will add or subtract the backlash distance from the motor's commanded position during all subsequent moves. If the motor moves in a positive direction, the distance will be added; if the motor moves in a negative direction, it will be subtracted. When the motor finishes a move, it will remain in the compensated position until the next move.

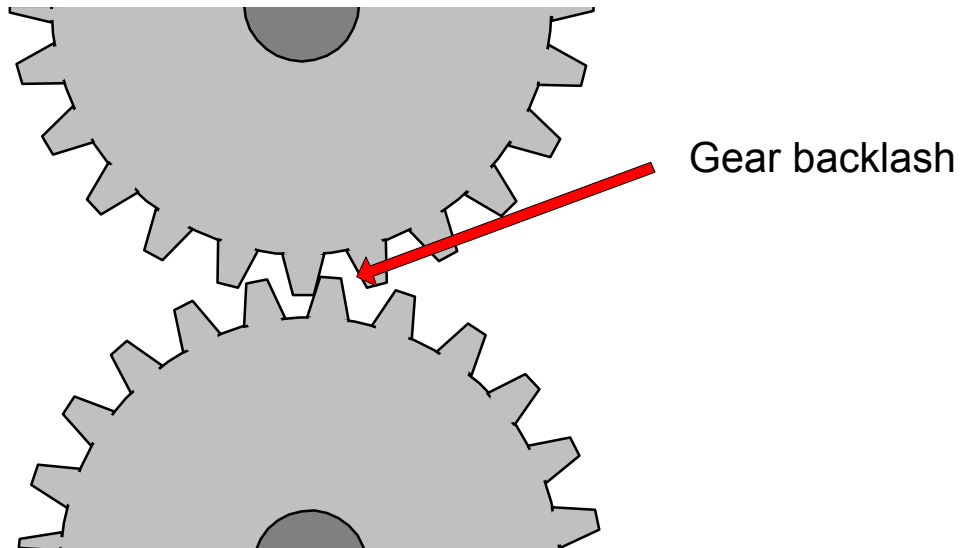
Prior to enabling backlash compensation, the motor should be positioned halfway between the two positions where it makes contact with the mechanical gearing. This will allow the controller to take up the backlash when the first move in either direction is made, without "bumping" the mechanical position.

While backlash compensation is enabled, the response to the **MCGetPosition( )**, **MCTellTarget( )** and **MCTellOptimal( )** commands will be adjusted to reflect the ideal positions as if no mechanical backlash was present.

For the example below assume that the system has 200 encoder counts of backlash. This example moves the system to the middle of the backlash range and enables compensation. Note that the

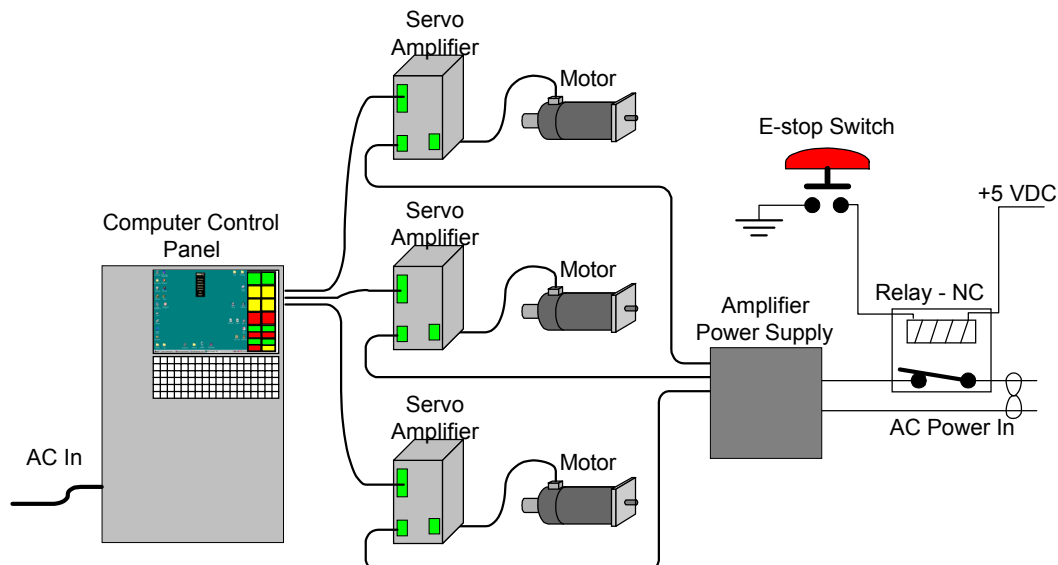
compensation value (in encoder counts) used with **MCEnableBacklash()** is half of the total amount of backlash.

```
MCMoveRelative( hCtrlr, 1, -100.0 );           // move to middle of backlash
MCWaitForStop( hCtrlr, 1 );                   // let motion finish
MCEnableBacklash( hCtrlr, 1, 100.0, TRUE );    // enable backlash compensation
```



## 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.

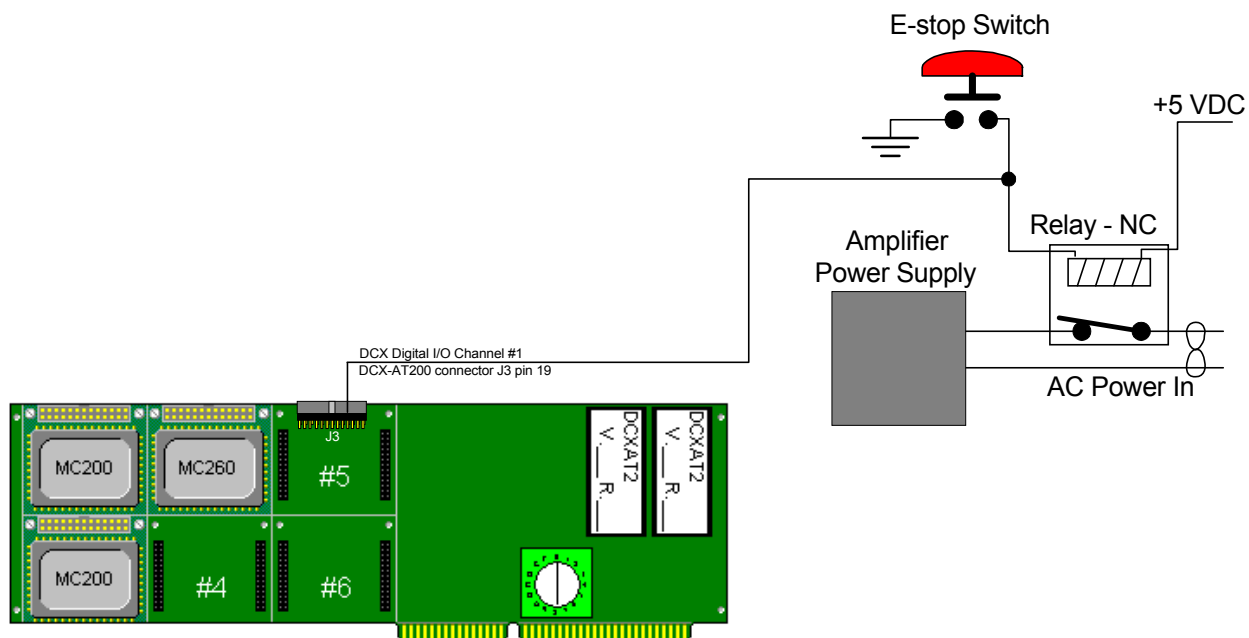
### Wiring the E-Stop switch to the DCX

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 (MC200 & MC210) 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 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 ( hCtrl, MC_BLOCK_TASK, 0 ) == MCERR_NOERROR ) {
    MCSetRegister ( hCtrl, 100, 0, MC_TYPE_LONG);
    MCConfigureDigitalIO ( hCtrl, 1, MC_DIO_LOW );
    MCWaitForDigitalIO ( hCtrl, 1, TRUE );
    MCSetRegister hCtrl, 100, 1, MC_TYPE_LONG);
    MCEnableAxes ( hCtrl, MC_ALL_AXES, FALSE );
    MCBlockEnd ( hCtrl, 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 (hCtrl, 100, &Estop, MC_TYPE_LONG);

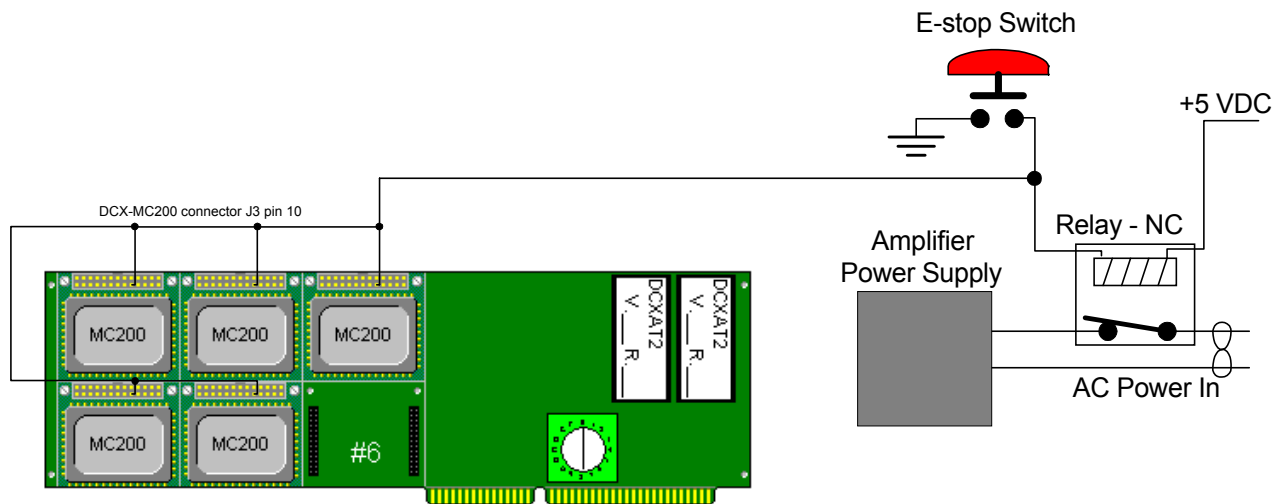
```

### 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 32 bit position resolution, resulting in a position range of  $-2,147,483,647$  to  $2,147,483,647$ . For an application where the axis is moving at maximum velocity (1million encoder counts/steps per second), the encoder would rollover in approximately 35.8 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 does not support executing Position Mode moves when the encoder rolls over. No matter what the commanded position, the axis will stop at the rollover position ( $2,147,483,647$  or  $-214,748.3647$ ).

### 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.



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 an offset to the home position will cause the axis to pause at the rollover point.

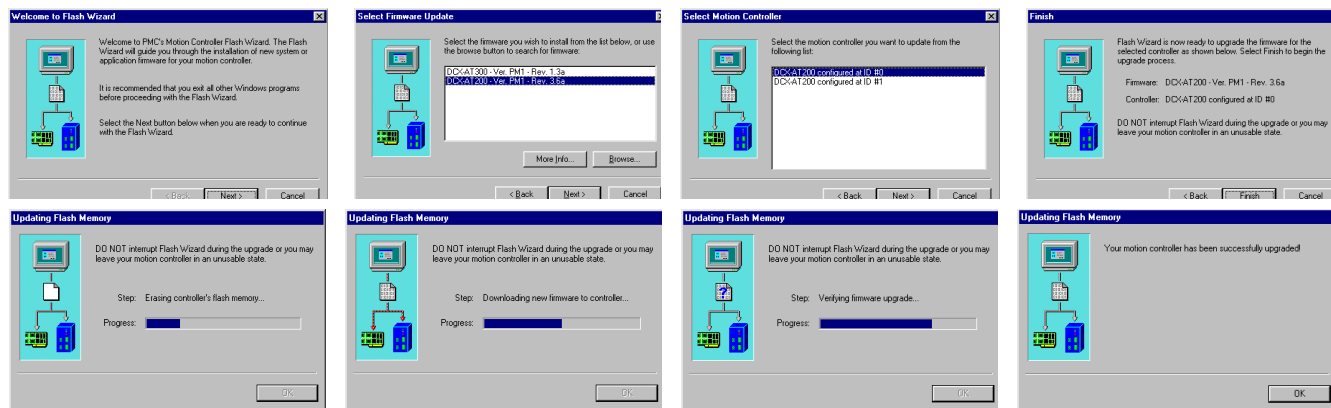
## Flash Memory Firmware Upgrade

The DCX firmware is stored in Non-Volatile FLASH memory. This allows the machine builder/user to easily upgrade to the most current revision of code, available from the PMC web site [www.pmccorp.com/support/DCX-AT200](http://www.pmccorp.com/support/DCX-AT200).

There are two methods for upgrading the firmware. The PMC Flash Wizard requires no jumper changes on the DCX and supports Windows 95/98 and NT. The batch file method requires jumper changes but will upgrade the firmware even if the firmware has been corrupted. The batch file upgrade supports DOS, Windows 3.X, 95/98 **but does not support NT**.

### Flash Wizard

The eight Flash Wizard windows are shown below. Firmware revision 3.6A or higher is required for Flash Wizard firmware upgrades. If this upgrade fails, the batch file upgrade procedure must be used.



### Batch file upgrade

Step 1) Change the jumper settings of the DCX-AT200:

JP2: 1 to 2, 3 to 4, and 5 to 6     **closed**  
 JP3: 1 to 2     **open**  
 JP4: 1 to 2     **closed**

Step #2) Memory address switch SW1 **must be set to position 0**

Step #3) When power is applied, all red LED's will be on. To begin the download run the firmware

download batch file: flash.bat

Step #4) When the following message is displayed enter carriage return:

Release the DCX-AT100 reset switch, press any key

Step #5) After the previous firmware version is erased (approximately 15 seconds) the file download will begin. Axis #1 and Axis #2 error LED's will strobe. Depending on the speed of the PC computer the download will take about 20 seconds

The batch file upgrade does not support Windows NT systems. The DCX will need to be reprogrammed in a Windows 3.X, 95/98 or DOS computer. The batch file upgrade does not support reprogramming a DCX at addresses other than D000:0000.

## Laser Cutting

For laser cutting applications, the DCX can directly control the power of a laser. While other DCX-MC2XX motion control modules are controlling the motion along the desired path, an additional MC200 module is used to generate the unipolar, TTL level, PWM output signal that controls the output of the laser. In this mode of operation the Direction output of the MC200, available on connector J3 pin 7, is re-defined as PWM output. The signal defined as Analog Output (J3 pin 2) is used for the direction signal.

To enable the PWM output, the **MCSetOOutputMode( )** command is issued with the *wMode* parameter set to **MC\_OM\_UNI\_PWM**. The motor module will now output a PWM signal at a frequency of 1.4648 KHz. The PWM characteristics that must be defined by the user are Minimum Duty Cycle and Maximum Duty Cycle.



The current release of the Motion Control API (2.20.0000) does not provide a high level function calls that defines the Output Deadband of an axis. The following description uses the MCAPI OEM low level function **pmccmdex( )** to issue the MCCL command Output Deadband (**aODn**) which is used to configure an axis for laser control.

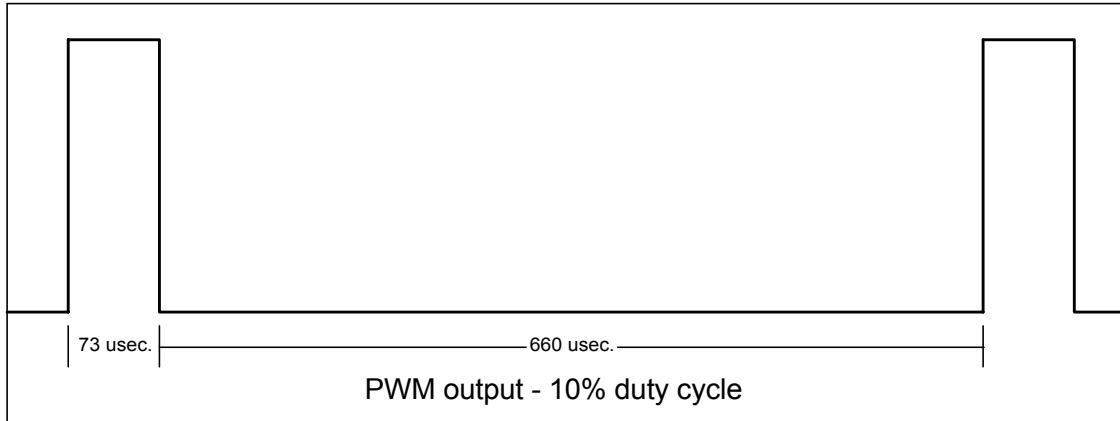
Future releases of the MCAPI will resolve this lack of support.

Minimum duty cycle defines the least amount of power that will be applied to the laser during a contour move. This value is typically specified as a percentage of the frequency of the PWM. The Minimum Duty Cycle (MinDC) is programmed using the Output Dead band command (**aODn**) where *a* is the axis number of the PWM module and *n* is the scaled percentage of the PWM frequency. The following calculation is used to determine the value *n* for a minimum duty cycle of 10%:

$$\begin{aligned} n &= \text{PWM constant} * \text{desired minimum duty cycle} \\ n &= 10 * 10\% \\ n &= 10 * .1 \\ n &= 1 \\ \text{MinDC} &= \text{aOD1} \end{aligned}$$

Based on the PWM frequency of 1.4648 KHz, the Minimum Duty Cycle Period (MDCP) will be:

$$\begin{aligned}
 \text{MDCP} &= (\text{PWM Frequency}/2) * 10\% \\
 &= (1.4648/2) * 10\% \\
 &= 732.4 \text{ usec.} * 10\% \\
 &= 732.4 \text{ usec.} * .1 \\
 &= 73.24 \text{ usec.} \\
 \text{MDCP} &= 73 \text{ usec. (rounded)}
 \end{aligned}$$

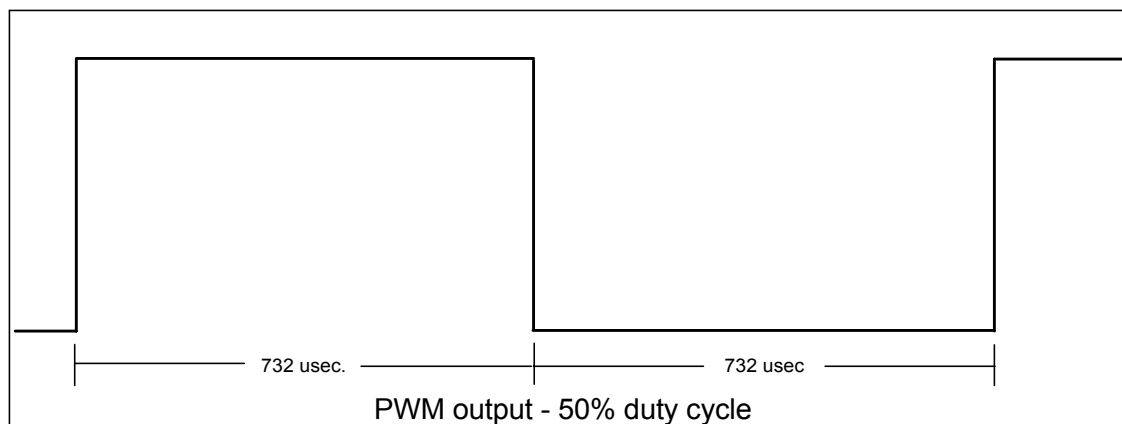


Selection of the Maximum Duty Cycle will be based on the specifications of the laser and the application specifics (motors, mechanics, thickness of material to be cut, etc...). This setting will determine the maximum power that will be applied to the laser. For this application example, the maximum duty cycle will be 50%. The Maximum Duty Cycle (MaxDC) is set by combining the Minimum Duty Cycle (MinDC) and the Remaining Duty Cycle (RDC).

$$\begin{aligned}
 \text{MaxDC} &= \text{MinDC} + \text{RDC} \\
 \text{RDC} &= \text{MaxDC} - \text{MinDC} \\
 \text{RDC} &= 50\% - 10\% \\
 \text{RDC} &= 40\%
 \end{aligned}$$

The **VelocityGain** member of the **MCFilter** data structure is used to define the Remaining Pulse Period (RPP). For a 50% maximum duty cycle at a maximum vector velocity of 10,000 encoder counts per second, the following calculation is used to determine the Velocity Gain parameter  $n$  :

$$\begin{aligned}
 n &= (\text{PWM Constant} * \text{RDC}) / \text{Maximum encoder Counts per second} \\
 n &= (10 * 40\%) / 10,000 \\
 n &= (10 * .4) / 10,000 \\
 n &= 4 / 10000 \\
 \text{aVGn} &= .0004
 \end{aligned}$$



In the following example, axes 1 and 2 are used to draw a triangle. Axis 3 is slaved to axis 1 (Contour move profiling axis) and will generate the PWM output signal with a frequency of 1.464 KHz. The resolution of the PWM is 16 bit. The PID loop gains (proportional, derivative, and integral) for axis 3 are set to 0. For this application the duty cycle of the PWM output must range from a minimum of 10% to 50%. The maximum vector velocity of the X and Y axes motion is 10,000 encoder counts per second.

```
MCSetOperatingMode( hCtrlr, 1, 1, MC_MODE_CONTOUR ); // axis 1 contour mode
MCSetOperatingMode( hCtrlr, 2, 1, MC_MODE_CONTOUR ); // axis 2 contour mode

MCGetContourConfig( hCtrlr, 1, &Contour );
Contour.VectorVelocity = 10000.0 )
Contour.VectorAccel = 10000.0 )
Contour.VectorDecel = 10000.0 )
MCSetContourConfig( hCtrlr, 1, &Contour );

MCSetModuleOutputMode( hCtrlr, 3, MC_OM_UNI_PWM ); //PWM output mode

MCSetGain( hCtrlr, 3, 0.0 );
MCGetFilterConfig( hCtrlr, 3, &Filter );
Filter.DerivativeGain = 0;
Filter.IntegralGain = 0;
Filter.FollowingError = 0;
MCSetFilterConfig( hCtrlr, 3, &Filter );

// Use the MCCL command Output Deadband to define the minimum duty cycle. For this
// calculation the value of Maximum Duty Cycle Constant = 10. 10% duty cycle
// is 10*10%=1.0. Header file MCAPI.H must be included
//
if (pmcrdy( hCtrlr )) {
    arg = 1.0;
    if (pmccmdex( hCtrlr, 3, OD, &arg, MC_TYPE_DOUBLE ) == MCERR_NOERROR) {
    }
}

// Use velocity gain + minimum duty cycle to define the maximum duty cycle.
//
MCGetFilterConfig( hCtrlr, 3, &Filter );
Filter.VelocityGain = 0.0004;
MCSetFilterConfig( hCtrlr, 3, &Filter );
MCEnableAxis( hCtrlr, 3, TRUE );
```



---

```
// Set up axis 3 as slave to the controlling axis of the contour group
//
MCEnableGearing( hCtrlr, 3, 1, 1.0, TRUE );

// Linear move, first side of triangle
//
MCBlockBegin( hCtrlr, MC_BLOCK_CONTR_LIN, 1 );
    MCMoveAbsolute( hCtrlr, 1, 10000.0 );
    MCMoveAbsolute( hCtrlr, 2, 10000.0 );
MCBlockEnd( hCtrlr, NULL );

// Linear move, second side of triangle
//
MCBlockBegin( hCtrlr, MC_BLOCK_CONTR_LIN, 1 );
    MCMoveAbsolute( hCtrlr, 1, 200000.0 );
    MCMoveAbsolute( hCtrlr, 2, 0.0 );
MCBlockEnd( hCtrlr, NULL );

// Linear move, third side of triangle
//
MCBlockBegin( hCtrlr, MC_BLOCK_CONTR_LIN, 1 );
    MCMoveAbsolute( hCtrlr, 1, 0.0 );
    MCMoveAbsolute( hCtrlr, 2, 0.0 );
MCBlockEnd( hCtrlr, NULL );

// Disable gearing of axis 3
//
MCEnableGearing( hCtrlr, 3, 1, 1.0, FALSE );

MCEnableAxis( hCtrlr, 3, FALSE );
```

## 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. For some applications, using **MCLearnPoint( )** to load a series of moves may be 'easier' than issuing a series of contour mode linear moves, even though the results would be the same.

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( hCtrlr, 1, TRUE );           // motor on
MCGoHome( hCtrlr, 1 );                     // start from absolute zero
MCWaitForStop( hCtrlr, 1, 0.100 );
```

```

for (wIndex = 0; wIndex < 5; wIndex++) {
    MCMoveRelative( hCtrlr, 1, 1234.0 ); // move
    MCWaitForStop( hCtrlr, 1, 0.100 ); // are we there yet?
    MCLearnPoint( hCtrlr, 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( hCtrlr, 4, FALSE ); // motor off
for (wIndex = 0; wIndex < 5; wIndex++) {
    MCMoveRelative( hCtrlr, 4, 2468.0 ); // nothing actually moves
    MCLearnTarget( hCtrlr, 4, wIndex, MC_LRN_TARGET );
}

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

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

```

To cause the DCX to perform linear interpolated moves between the taught points, place each of the axes in contour mode. Use the lowest axis number as the contour mode command parameters, this is the controlling axis. Set the vector velocity and accelerations of the controlling axis. Issue a single **MCMoveToPoint()** command to the controlling axis with the point numbers as the command parameter. Note that when point memory is used with motors in contour mode, point 0 should not be used. This example executes linearly interpolated moves through three stored points of axes 1, 2, and 3.

```

MCSetOperatingMode( hCtrlr, 1, 1, MC_MODE_CONTOUR );
MCSetOperatingMode( hCtrlr, 2, 1, MC_MODE_CONTOUR );
MCSetOperatingMode( hCtrlr, 3, 1, MC_MODE_CONTOUR );

// Linear interpolated move sequence through stored points

for (wIndex = 1; wIndex < 4; wIndex++) {
    MCBlockBegin( hCtrlr, MC_BLOCK_CONTR_LIN, 1 );
    MCMoveToPoint( hCtrlr, 1, wIndex );
    MCMoveToPoint( hCtrlr, 1, wIndex );
    MCMoveToPoint( hCtrlr, 1, wIndex );
    MCBlockEnd( hCtrlr, NULL );
}

```

## Record Motion Data

The DCX supports capturing and retrieving motion data for servo axes (MC200, MC210). 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. The MCAPI function **MCaptureData()** is used to acquire motion data for a servo axis. This function supports capturing:

- Actual Position versus time
- Optimal Position versus time
- Following error versus time
- DAC output versus time (DCX-MC200 only)

The time base (2 KHz, 1 KHz, 0.5 KHz) for captured data is set by **Rate** member of the **MCMotion** data structure. The function **MCGetCapturedData()** is used to retrieve the captured data. This example captures 1000 data points from axis 3, then reads the captured data into an array for further processing.

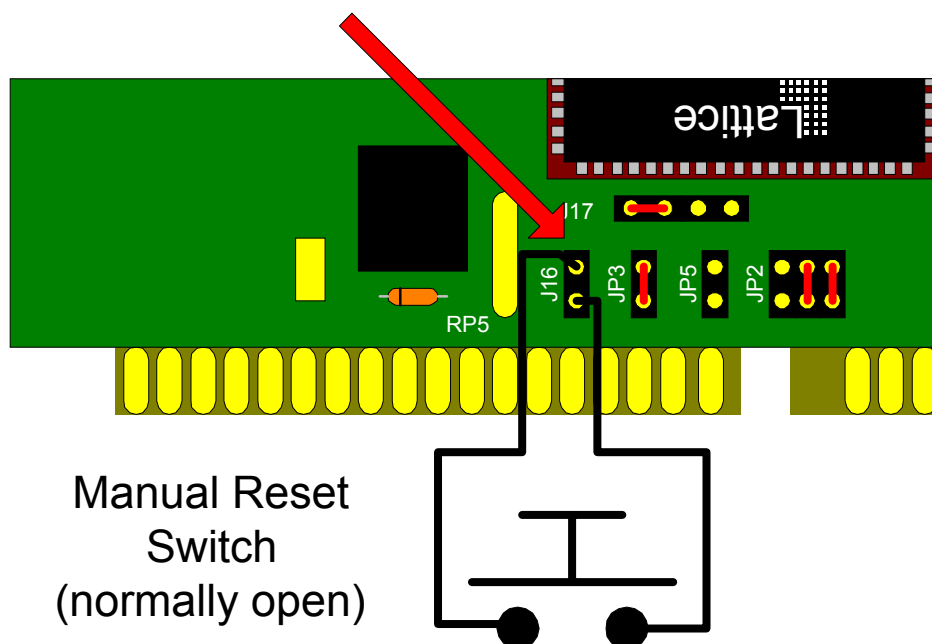
```
double Data[1000];

MCBlockBegin( hCtrlr, MC_BLOCK_COMPOUND, 0 );
MCCaptureData( hCtrlr, 3, 1000, 0.001, 0.0 );
MCMoveRelative( hCtrlr, 3, 1000.0 );
MCWaitForStop( hCtrlr, 3, 0.0 );
MCBlockEnd( hCtrlr, NULL );

// Retrieve captured actual position data into local array
//
if (MCGetCaptureData( hCtrlr, 3, MC_DATA_ACTUAL, 0, 1000, &Data ) {
    . . .          // process data
}
```

## Manually Resetting the DCX

The DCX supports using a switch to manually reset the card at anytime. Connect a normally open switch to pins one and two of jumper J16. Activating the switch will assert the internal reset signal of the DCX.





In the event of a 'system hang up', even though the DCX controller has been reset, the application program may fail to execute properly due to PC computer/DCX driver issues. If this is the case it will be necessary to reboot the computer system.

## Tangential Knife Control

A variation of Master/Slave mode supports using the position of two master axes to control the position of a third axis. The slave's optimal position will equal the arctangent of the ratio of the master axes' velocities. If the master axes are driving an X-Y table, the slave's position will equal the table's direction of travel. This dual master capability can be used to control the knife in cutting applications. This function is only available when the slave is a servo, and the two master axes, which can be servos or steppers, are in contour mode.



The current release of the Motion Control API (2.20.0000) does not provide a high level function call that enables tangential knife control. The following description uses the MCAPI OEM low level function ***pmccmdex()*** to issue the MCCL command Set Master (***aSMn***) with a parameter ***n*** which configures the axis that controls the rotation of the knife.

Future releases of the MCAPI will resolve this lack of support.

Set the scaling of the knife axis to one unit equals 360 degrees of rotation of the knife. Issue the Set Master (***aSMn***) command to the slave axis with a parameter ***n*** that specifies the two master axes. The value of the Set Master parameter should be calculated as follows:

parameter ***n*** = master 1 axis number + (master 2 axis number x 16)

With two master operation, the slave axis will begin to track the master axis's direction when the first (and subsequent) contour mode move is issued. The blade of the knife will remain tangential to the contour path. To terminate the master and slave connections between the axes, issue the Set Master command to the slave axis with a parameter of 0, followed by either the Position Mode (PM) or the Velocity Mode (VM) command. If a significant change in direction (like a corner) of the X and/or Y axes occurs the knife will instantaneously. If this is undesirable, lift the blade, place the slave in position mode, re-position the blade, and lower the blade.

The following example will cut a 5 inch square out of a piece of linoleum. Axes 1 and 2 (X and Y respectively) are designated as the two master axes. Axis 3 will position the knife. Axis four (Z) is used to lift the knife at a corner, where an instantaneous change of direction in X and/or Y would be undesirable.

```
// define scaling of axis 3, 2000 encoder counts per revolution sets 1 unit to
// 1 ;revolution
//
MCGetScale( hCtrlr, 3, &Scaling );
Scaling.Scale = 2000.0;
MCSetScale( hCtrlr, 3, &Scaling );
```

```

// Use the MCCL command Set Master to configure axis 3 as a slave to axes 1 and 2.
// Header file MCAPI.H must be included
//
if (pmcrdy( hCtrlr )) {
    arg = 33;
    if (pmccmdex( hCtrlr, 3, SM, &arg, MC_TYPE_LONG ) == MCERR_NOERROR) {
    }
}

// turn on axes 1, 2, 3, & 4
//
MCEnableAxis( hCtrlr, 1, MC_ALL_AXES );

//Execute 1st linear move
//
MCSetOperatingMode( hCtrlr, 1, 1, MC_MODE_CONTOUR ); // axis 1 contour mode

// Linear move, first side of triangle
//
MCBlockBegin( hCtrlr, MC_BLOCK_CONTR_LIN, 1 );
    MCMoveAbsolute( hCtrlr, 1, 1000.0 );
    MCMoveAbsolute( hCtrlr, 2, 0.0 );
MCBlockEnd( hCtrlr, NULL );

// wait for end of contour move, lift blade, rotate blade, lower blade
//
MCWaitForStop( hCtrlr, 1, 0.1 );
MCMoveRelative( hCtrlr, 4, 1000.0 );
MCWaitForStop( hCtrlr, 4, 0.1 );
MCMoveRelative( hCtrlr, 3, 0.333 );
MCWaitForStop( hCtrlr, 3, 0.1 );
MCMoveRelative( hCtrlr, 4, -1000.0 );
MCWaitForStop( hCtrlr, 4, 0.1 );

// Linear move, second side of triangle
//
MCBlockBegin( hCtrlr, MC_BLOCK_CONTR_LIN, 1 );
    MCMoveAbsolute( hCtrlr, 1, 500.0 );
    MCMoveAbsolute( hCtrlr, 2, 1000.0 );
MCBlockEnd( hCtrlr, NULL );

// wait for end of contour move, lift blade, rotate blade, lower blade
//
MCWaitForStop( hCtrlr, 1, 0.1 );
MCMoveRelative( hCtrlr, 4, 1000.0 );
MCWaitForStop( hCtrlr, 4, 0.1 );
MCMoveRelative( hCtrlr, 3, 0.333 );
MCWaitForStop( hCtrlr, 3, 0.1 );
MCMoveRelative( hCtrlr, 4, -1000.0 );
MCWaitForStop( hCtrlr, 4, 0.1 );

// Linear move, third side of triangle
//
MCBlockBegin( hCtrlr, MC_BLOCK_CONTR_LIN, 1 );
    MCMoveAbsolute( hCtrlr, 1, 0.0 );
    MCMoveAbsolute( hCtrlr, 2, 0.0 );
MCBlockEnd( hCtrlr, NULL );

// wait for end of contour move, lift blade, rotate blade, lower blade

```

```
//
MCWaitForStop( hCtrlr, 1, 0.1 );
MCMoveRelative( hCtrlr, 4, 1000.0 );
MCWaitForStop( hCtrlr, 4, 0.1 );
MCMoveRelative( hCtrlr, 3, 0.333 );
MCWaitForStop( hCtrlr, 3, 0.1 );
MCMoveRelative( hCtrlr, 4, -1000.0 );
MCWaitForStop( hCtrlr, 4, 0.1 );

!!! now disable tangential knife control !!!
```

## Threading Operations

Threading operations require not only tight synchronization between the primary axes, but also the ability to begin motion of the slave axis relative to a specific position of the master. The DCX implementation of threading uses the encoder index mark of the master axis to trigger motion of the slave.



The current release of the Motion Control API (2.20.0000) does not provide a high level function call for enabling threading operations. The following description uses the MCAPI OEM low level function ***pmccmdex()*** to issue the MCCL command Set Master (***aSMn***) with a parameter ***n*** which configures the DCX controller for threading.

Future releases of the MCAPI will resolve this lack of support.

To enable Master/Slave Threading mode, issue the Set Master (***aSMn***) command where:

a = the axis number of the slave  
n = the axis number of the master + 2

A move absolute, move relative, or go home command can also be issued to the slave axis to set a target position where the axis will be taken out of slave mode. The Index Arm or Find Index command must be issued to the master axis after the Set Master command has been issued to the slave axis. The slave will be synchronized to the master's position when its encoder index pulse occurs. In the following example the spindle (master) is axis #2 and the thread cutting tool is positioned by axis #1 (slave).

```
// Set scaling of master axis. For the spindle, this would typically be set to
// the number of encoder counts per revolution.
//

MCGetScale( hCtrlr, 2, &Scaling );
Scaling.Scale = 2000.0;
MCSetScale( hCtrlr, 2, &Scaling );
```

---

```

//Set scaling of the slave axis
//
MCGetScale( hCtrlr, 1, &Scaling );
Scaling.Scale = 4000.0;
MCSetScale( hCtrlr, 1, &Scaling );

MCMoveAbsolute( hCtrlr, 1, 0.0 );           // move slave to starting position

MCWaitForStop( hCtrlr, 1, 0.1 );           // wait till we're there

// Set the slave ratio. This is the lead or pitch when cutting a thread.
//
MCEnableGearing( hCtrlr, 1, 2, 0.1, TRUE );

// Use the MCCL command Set Master to configure axis 2 as a slave to axis 1.
// Enable threading by n = 2 + 256. Header file MCAPI.H must be included
//
if (pmcrdy( hCtrlr )) {
    arg = 258;
    if (pmccmdex( hCtrlr, 3, SM, &arg, MC_TYPE_LONG ) == MCERR_NOERROR) {
    }
}

// Set the target position. This is the position at which slave mode is
// terminated and axis #1 will stop.
//
MCMoveAbsolute( hCtrlr, 1, 1.0 );

// Start master axis moving in torque mode.
//
MCSetTorque( hCtrlr, 2, 3.0 );

// Arm the index capture of the master axis. When the index pulse occurs, the
// slave will begin tracking the master axis until // the slave reaches its
// target position.
2IA

// This command sequence will repeat until auxiliary status bit 22 is clear,
// indicating that the slave has reached its target.
//
1RL16,IS22,JR-2,NO,2SQ0

```

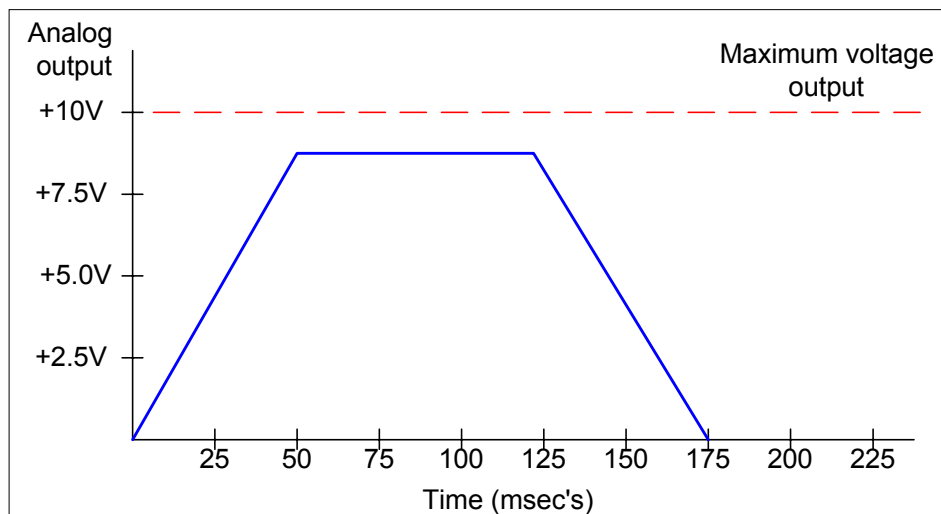
The following bits of the axis auxiliary status word are used for monitoring the status of the slave axis during a threading operation:

- Bit 22 = Axis is slaved to master's encoder position
- Bit 23 = Axis is slaved and waiting for master's index mark

## Torque Mode Output Control

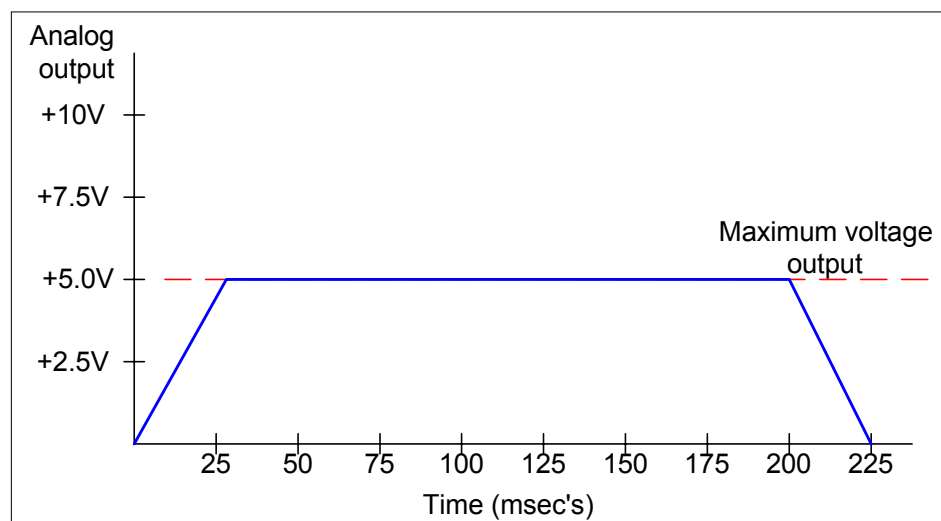
The DCX servo modules (MC200 & MC210) provide two methods of **directly and completely** controlling the Torque/Velocity of a axis. When executing closed loop servo motion in Position or Velocity mode, the **MCSetTorque( )** command allows the user to limit the output signal or duty cycle

to a specific level. The following graph depicts a simple position mode move of 1000 encoder counts with the default torque setting of 10 volts (no limit).



The graphic below depicts the same 1000 encoder count move, but the maximum voltage output has been limited to 5.0 volts.

```
MCSetTorque( hCtrlr, 1, 5.0 );
MCMoveRelative( hCtrlr, 1, 1000.0 );
```



### Servo Modules as simple D/A or PWM output with encoder reader

Selecting Torque mode using the **MCSetOperatingMode()** function allows the user to directly write values to the servo control DAC. This mode does not support closed loop servo control, but the user can read the position of the encoder at any time.

```
MCSetOperatingMode( hCtrlr, 1, 0, MC_MODE_VELOCITY );
MCSetTorque( hCtrlr, 1, 2.5 );           ;axis 1 output to 2.5V (MC200)
MCSetTorque( hCtrlr, 1, 7.5 );           ;set duty cycle to 75% (MC210)
```



## Defining User Units

When power is applied or the DCX is reset, it defaults to encoder counts or stepper pulses 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. If the user issues the same command to a stepper motor, it will move 1000 motor steps.

In many applications there is a more convenient unit of measure than the encoder counts of the servo or steps of the stepper motor. If there is a fixed ratio between the encoder counts or steps 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/Step) Units

The value of the **Scale** member is the number of encoder counts or steps 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( hCtrlr, 3, &Scaling );
Scaling.Scale = 1000.0;           // 1000 encoder counts/inch
MCSetScale( hCtrlr, 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, and deceleration) but not PID settings.

### Trajectory Time Base

The value of the **Rate** member sets the time unit for velocity, acceleration and 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, acceleration and deceleration 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( hCtrlr, 3, &Scaling );
Scaling.Rate = 60.0;                // set rate to inches per minute
MCSetScale( hCtrlr, 3, &Scaling );
```

#### Typical Rate values

<b>Time Unit</b>	<b>User Rate Conversion</b>
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( hCtrlr, &Scaling );
Scaling.Time = 60.0;                // set Wait time unit to minutes
MCSetScale( hCtrlr, &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 or stepper 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 or stepper motor, it only establishes an arbitrary zero position for the axis.

```
MCSCALE Scaling;

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

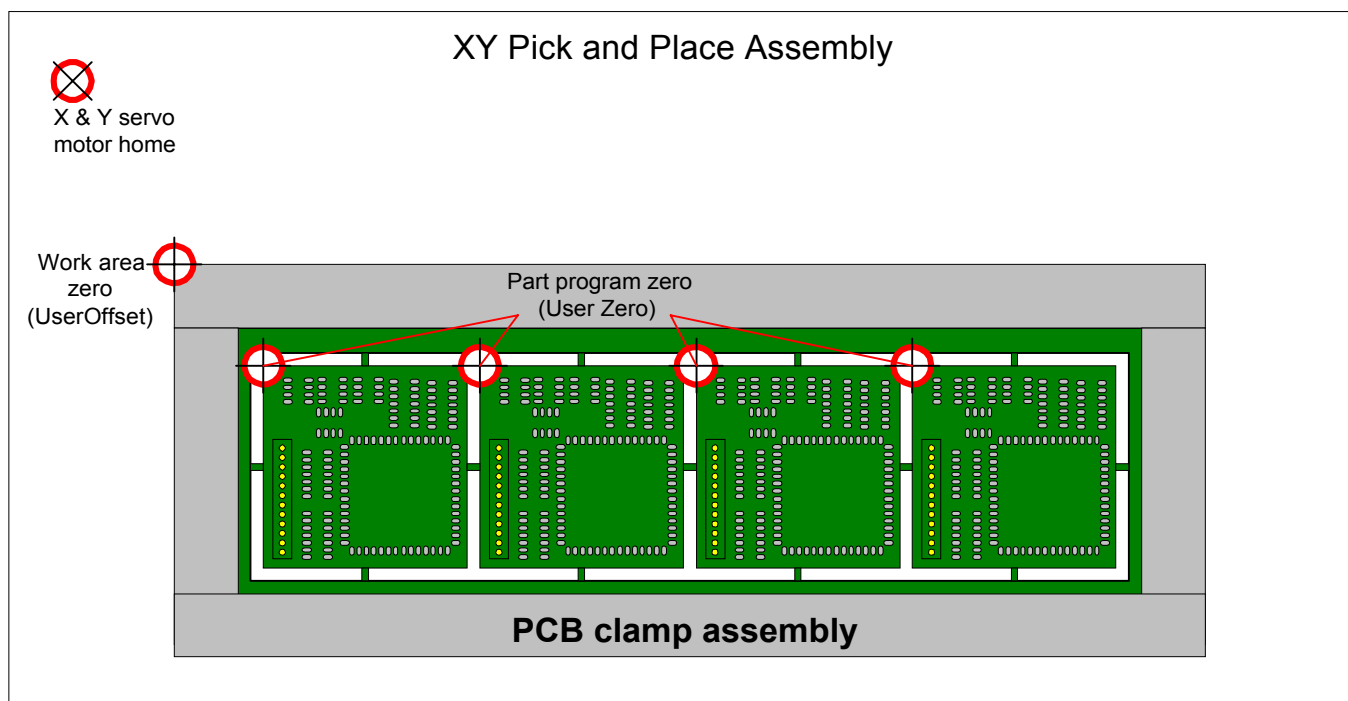
### Defining a Part Zero

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



### Defining the output constant for velocity gain

The member **Constant** allows the user to define the units to be used for setting the Velocity Gain parameters. Please refer to the description of **Using Velocity Gain** in the **Application Solutions** chapter of this user manual.

## DCX Watchdog

The DCX incorporates a watchdog circuit to protect against improper CPU operation. If the DCX processor fails to properly execute firmware code for a period of 100 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 outputs (+/- 10V, PWM, Step/Direction) will be disabled. When the watchdog circuit has tripped, the two yellow LEDs (L2 and L3) will be turned on. To clear the watchdog error either:

Cycle power to the computer (**recommended**)  
Reset the computer  
Manually reset the DCX



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

---

- DCX Motherboard Digital I/O
- Configuring the DCX Digital I/O
- Using the DCX Digital I/O
- DCX Motherboard Analog Inputs
- DCX Module Analog I/O
- Using the Analog I/O
- Calibrating the MC500/MC520 +/- 10V Analog Outputs

## General Purpose I/O

---

### DCX Motherboard Digital I/O

The DCX-AT200 Motion Controller motherboard has 16 undedicated digital I/O channels. These signals can be accessed on connector J3 of the motherboard. The **DCX-AT200** section of the **Connectors, Jumpers, and Schematics** chapter includes a pin-out for this connector. Each digital channel is configured via software (input, output, high true, low true).

#### Interfacing to the 'Outside World'

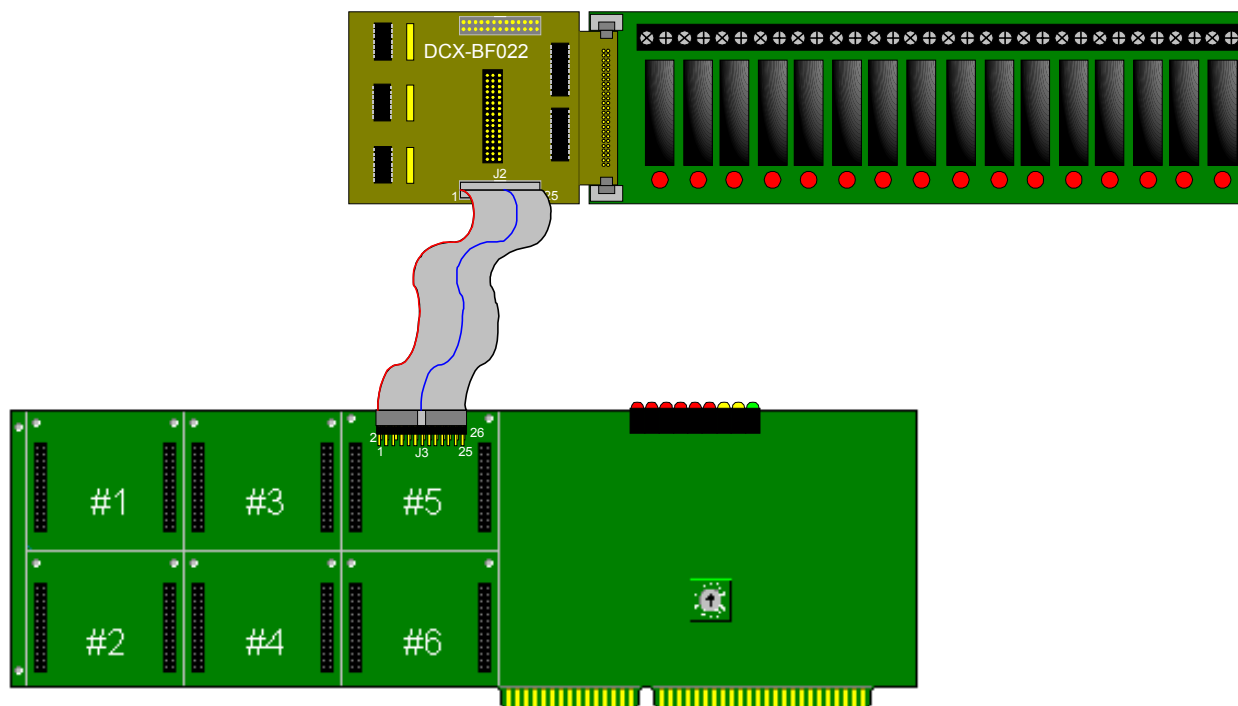
The TTL digital I/O channels can be connected directly to external circuits if output loading (**1ma maximum sink/source**) and input voltages (**0.0V to +5.0V**) are within acceptable limits.



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-AT200 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-BF022 plugs directly into the relay rack's 50 pin header connector and then connects to the DCX-AT200 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.

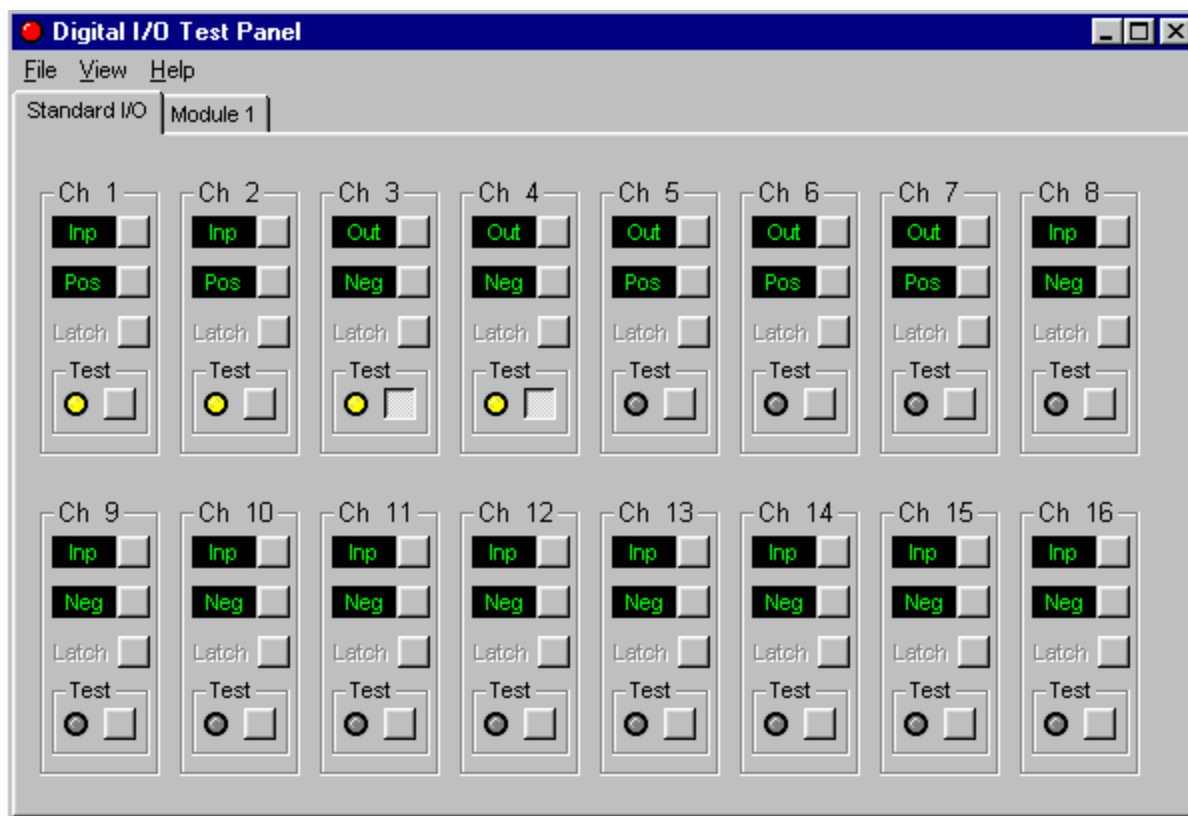
Although the relay rack has screw terminals for connecting a logic supply, it is not necessary to make this connection. By installing a shorting block on jumper JP17 of the BFO22, the 5 volt supply of the DCX will be supplied to the relay rack.

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.

## Configuring the DCX Digital I/O

The configuration of both the DCX-AT200 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 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-AT200 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 on a controller for output, then turns each channel on (in order) for a half second.

```
MCPARAM Param;

MCGetMotionConfig( hCtrlr, &Param );

for (i = 1; i <= Param.DigitalIO; i++) {
    MCConfigureDigitalIO( hCtrlr, i, MC_DIO_OUPUT | MC_DIO_HIGH );

for (i = 1; i <= Param.DigitalIO; i++) {
    MCEnableDigitalIO( hCtrlr, i, TRUE );
    MCWait( hCtrlr, 0.5 );
    MCEnableDigitalIO( hCtrlr, 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:

<b><i>MCEnableDigitalIO()</i></b>	set digital output channel state
<b><i>MCGetDigitalIO()</i></b>	get digital input channel state
<b><i>MCWaitForDigitalIO()</i></b>	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*.

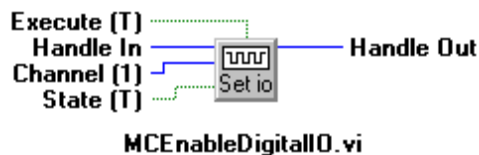
TRUE Turns the channel on.  
FALSE Turns the channel off.

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 hCtrlr, WORD wChannel, short int bState );  
**Delphi Function:** procedure MCEnableDigitalIO( hCtrlr: 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:**

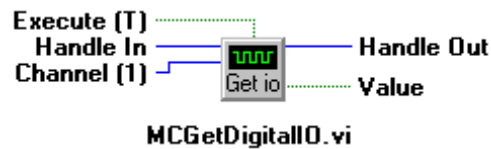
**C++ Function:** short int MCGetDigitalIO( HCTRLR hCtrlr, WORD wChannel );

**Delphi Function:** function MCGetDigitalIO( hCtrlr: 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

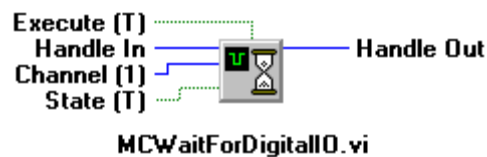
**C++ Function:** void MCWaitForDigitalIO( HCTRLR hCtrlr, WORD wChannel, short int bState );

**Delphi Function:** procedure MCWaitForDigitalIO( hCtrlr: 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.

```

MCPARAM Param;

MCGetMotionConfig( hCtrlr, &Param );

for ( i = 1; i <= Param.DigitalIO; i++) {
    MCConfigureDigitalIO( hCtrlr, i, MC_DIO_OUPUT | MC_DIO_HIGH );

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

```

```
//
// Next re-configure channel 3 for input, and put up a message
// box based on the input state
//
if (MCConfigureDigitalIO( hCtrlr, 3, MC_DIO_INPUT | MC_DIO_HIGH )) {
    val = MCGetDigitalIO( hCtrlr, 3 );
    if (val) // MessageBox is a Windows API function
        MessageBox( hParent, "Channel 3 input voltage high (>2.4VDC)",
                    "MCAPI Sample", MB_ICONINFORMATION );
    else
        MessageBox( hParent, "Channel 3 input voltage low (<0.4VDC)",
                    "MCAPI Sample", MB_ICONINFORMATION );
}
```

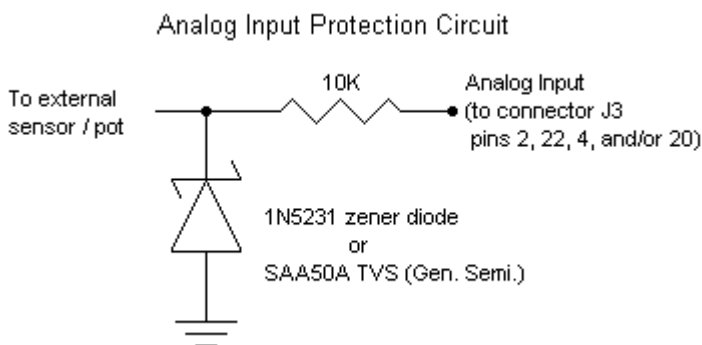
## DCX Motherboard Analog Inputs

The DCX-AT200 Motion Controller motherboard has 4 undedicated 8 bit analog input channels. These signals can be accessed on connector J3 of the motherboard. Appendix B includes a pin-out for this connector.

The analog input channels on the motherboard are numbered from 1 to 4. Each of these inputs can accept an analog input signal from 0 to +5 volts. To prevent damage to the DCX circuitry, the input signal must be limited to this range by external circuits.



A voltage level greater than 5.6 volts will damage DCX-AT200 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.



The DCX includes an on-board 5.00 volt reference supply for the analog to digital converter. This reference voltage will be used if a shorting block is installed on jumper JP1 of the DCX motherboard. In this case, the reference voltage will also be connected to pin 23 of motherboard connector J3 and can be used as an output to external components. Alternatively, an external reference voltage

between 0 and +5 volts can be connected to pin 23 of connector J3. In this case, the pins of jumper JP1 should be left open.

The DCX will perform a ratiometric conversion of the four input channels periodically. The result of each conversion will be a number between 0 and 255. If the on-board reference is used, the value will be the ratio of the input voltage to 5.0 volts, times 255. If an external reference voltage is used, the value will be the ratio of the input voltage to the reference voltage, times 255.

## DCX Module Analog I/O

The DCX-MC500 Analog I/O Module provides additional analog I/O capability to a DCX Motion Controller. One or more of these modules can be installed in any available 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:

<b>Part Number</b>	<b>Description</b>
DCX-MC500	4 Inputs and 4 Outputs
DCX-MC510	4 Inputs
DCX-MC520	4 Outputs

On each DCX-MC500 Analog I/O Module all analog input channels are numbered sequentially as a group. Likewise, all analog output channels are numbered sequentially as a group. When installed in the DCX-AT200, since there are already 4 analog input channels on the motherboard, the analog I/O channels on the DCX-MC500s start with number 5.

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. 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, which converts them to a corresponding analog output signal level.

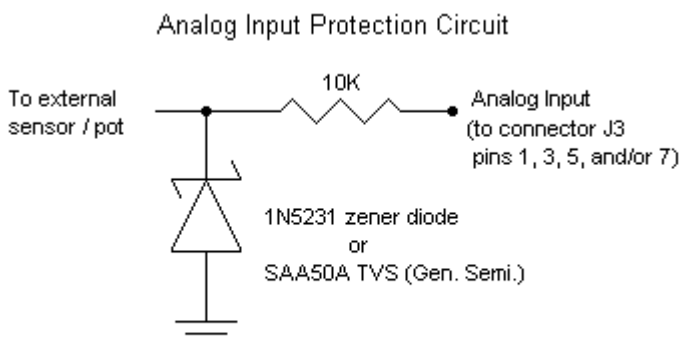
The DCX-MC500 is designed to accurately measure voltage levels on the input channels. These inputs are very high impedance with leakage currents less than 10 nano amps. The output channels are designed to provide signals with accurate voltage levels. The current requirement from these outputs should 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 DCX-AT200 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.



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.

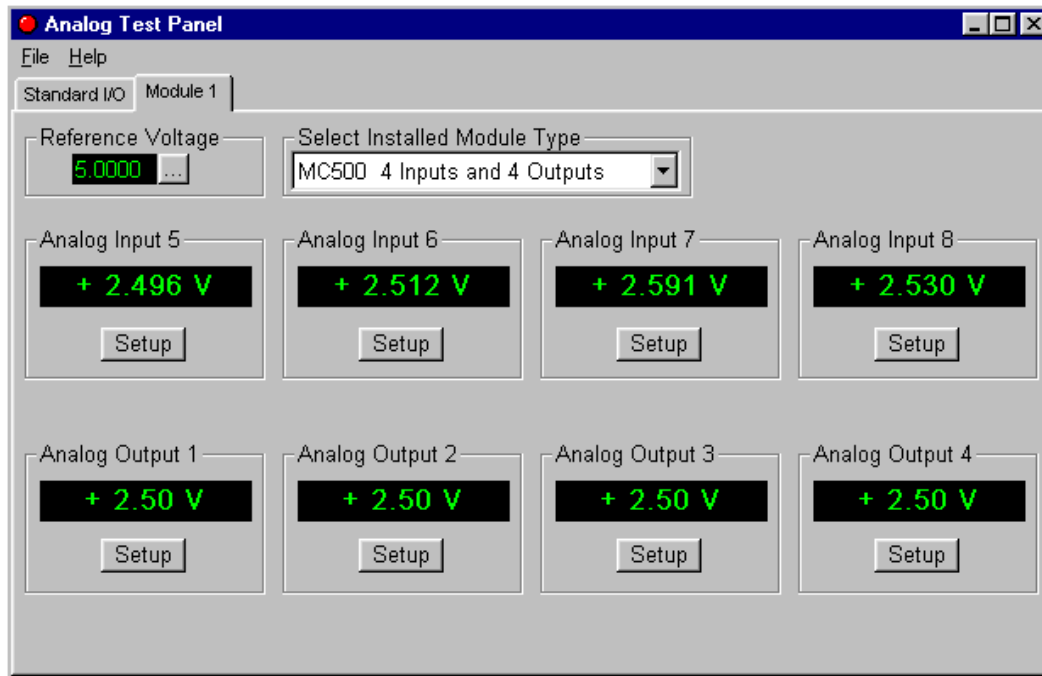
The outputs from the DAC on the DCX-MC500 module are voltage levels in the range 0 to +5 volts. These outputs have no gain or offset adjustment. These signals are available on pins 10, 12, 14 and 16 of the module J3 connector.

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 both the DCX-AT200 and 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

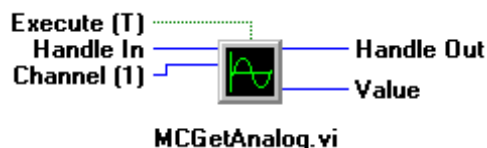
## Get Analog

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 hCtrlr, WORD wChannel );  
**Delphi Function:**     function MCGetAnalog( hCtrlr: 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

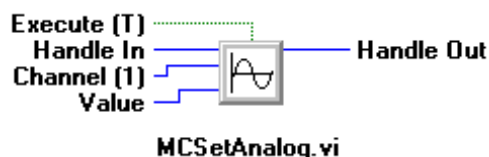
**C++ Function:** void MCSetAnalog( HCTRLR hCtrlr, WORD wChannel, WORD wValue );

**Delphi Function:** procedure MCSetAnalog( hCtrlr: HCTRLR; wChannel, value: Word );

**VB Function:** Sub MCSetAnalog (ByVal hCtrlr As Integer, ByVal channel As Integer, ByVal Value As Integer)

**MCCL command:** OA

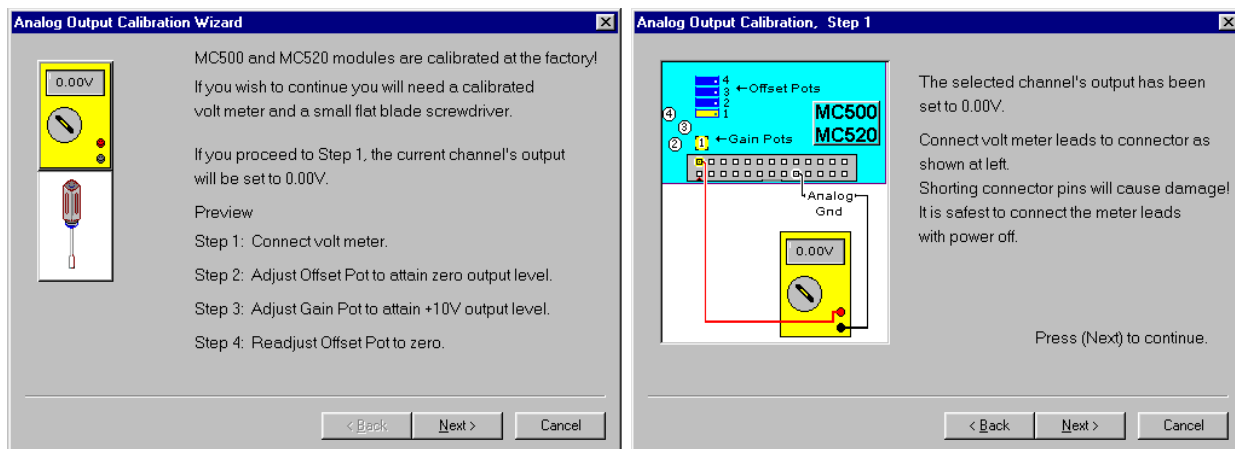
**LabVIEW VI:**



## 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 which 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**

---

- Introduction
- Motion Control API Function Quick Reference Tables
- Setup Functions
- Motion Functions
- Reporting Functions
- I/O Functions
- Macros and Multi-Tasking Functions
- MCAPI Driver Functions

## Motion Control API Function Reference

The Motion Control Application Programming Interface (API) implements a powerful set of high level functions and data structures for programming motion control applications. An example function description is shown below:

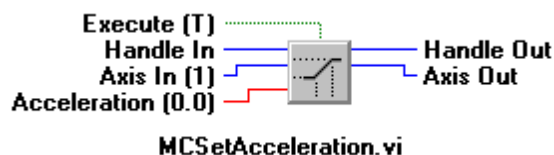
### Set Acceleration

Set the maximum acceleration rate for an axis. The default units for the command parameter are encoder counts (or steps) per second per second.

**compatibility:** MC200, MC210, MC260  
**see also:** Set Deceleration, Set Velocity, Set Motion Config

**C++ Function:** void MCSetAcceleration( HCTRLR hCtrlr, WORD wAxis, double Rate );  
**Delphi Function:** procedure MCSetAcceleration( hCtrlr: HCTRLR; wAxis: Word; Rate: Double );  
**VB Function:** Sub MCSetAcceleration (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal rte As Double)  
**MCCL command:** SA

**LabVIEW VI:**



The description (**Set Acceleration**) is derived from the operation that it performs

The "**compatibility**" line will list which DCX plug-in modules for which the function is valid. This is important since not all functions are supported on both servo and stepper axes. If a function isn't intended to be used for a specific axis or module, the letters 'N/A' for "Not Applicable" will appear on this line.

The "**see also**" line lists other MCAPI functions that have an association or similar purpose to the function being described.

The “**C++ Function:, Delphi Function:, and VB Function:**” lines show the high level function prototypes.

The “**MCCL command**” line lists the associated low level MCCL command.

Within each section (Setup, Motion, etc...) all functions that use data structures are listed first and all Data Structure members will be listed. The balance of the functions are listed alphabetically.

# Motion Control API Function Quick Reference Tables

## Setup Functions

<b>Function</b>	<b>Description</b>
MCSetAcceleration	set Acceleration for an axis
MCSetAuxEncPos	set the position of the auxiliary encoder
MCSetContourConfig	set contour configuration settings
MCSetDeceleration	Deceleration Set
MCSetFilterConfig	set the PID filter parameters
MCSetGain	set the proportional gain for a servo axis
MCSetLimits	configure hard and soft limits for an axis
MCSetModuleOutputMode	define the output type
MCSetMotionConfig	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
MCSetServoOutputPhase	select normal or reverse phasing for a servo axis
MCSetScale	set the scaling factors for an 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

## Motion Functions

<b>Function</b>	<b>Description</b>
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
MCContourPath	set the contour move type (linear, clockwise arc, counter clockwise arc)
MCDirection	set travel direction for velocity mode move
MCEnableAxis	turn axis on or off
MCEnableBacklashCompensation	enable backlash compensation
MCEnableJogging	enable/disable jogging
MCEnableGearing	enable/disable gearing
MCEnableSynch	enables cubic spline motion, synchronizes contour motion
MCFindAuxEnclDx	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
MCGo	start a velocity mode motion, begin cubic spline motion sequence
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
MCWaitForPosition	wait for axis to reach position
MCWaitForStop	wait for the calculated trajectory to be complete
MCWaitForTarget	wait for axis to reach target position

## Reporting Functions

<b>Function</b>	<b>Description</b>
MCDecodeStatus	axis status word decoding
MCErrNotify	enables/disables error messages for application window
MCGetAccelerationEx	get current programmed acceleration for axis
MCGetAuxEnclIdxEx	get last observed position of auxiliary encoder index pulse
MCGetAuxEncPosEx	get current position of auxiliary encoder
MCGetBreakpointEx	get the most recent breakpoint position
MCGetCapturedData	retrieve captured axis data (current position, optimal position, error)
MCGetContourConfig	get contour configuration settings
MCGetContouringCount	get current contour count
MCGetDecelerationEx	get current programmed deceleration for axis
MCGetFilterConfig	get the PID parameters
MCGetError	returns the most recent controller error
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
MCGetLimits	get current hard and soft limit settings
MCGetMotionConfig	get motion configuration
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
MCGetServoOutputPhase	get the output phase (normal or reversed) of a servo
MCGetScale	get the current programmed scaling factors for an axis
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
MCIsStopped	return value when trajectory is complete
MCIsAtTarget	return value when axis is at the target position
MCTranslateError	translate numeric error code to text message

## I/O Functions

<b>Function</b>	<b>Description</b>
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

<b>Function</b>	<b>Description</b>
MCCancelTask	cancel a background task
MCMacroCall	call a MCCL macro
MCRepeat	inserts a repeat command into a macro or task sequence

---

## MCAPI Driver Functions

<b>Function</b>	<b>Description</b>
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)
MCGetConfiguration	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

## Setup Commands

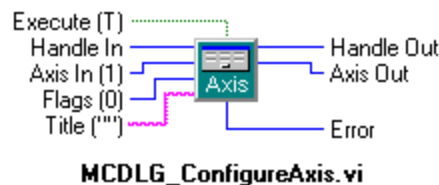
### Set Motion Configuration

This function provides a way of setting all motion parameters for a given *wAxis* with a single function call using an initialized **MCMOTION** structure. When you need to setup many parameters for an *wAxis* it is easier to call **MCGetMotionConfig()**, update the **MCMOTION** structure, and write the changes back using **MCSetMotionConfig()**, rather than using a Get/Set function call for each parameter. Note that some less often used parameters will only be accessible from this function and from **MCGetMotionConfig()** - they do not have individual Get/Set functions.

**compatibility:** MC200, MC210, MC260  
**see also:** Set Filter Configuration

**C++ Function:** short int MCSetMotionConfig( HCTRLR hCtrlr, WORD wAxis, MCMOTION far\* lpMotion );  
**Delphi Function:** function MCSetMotionConfig( hCtrlr: HCTRLR; wAxis: Word; var lpMotion: MCMOTION ): SmallInt;  
**VB Function:** Function MCSetMotionConfig ( ByVal hCtrlr As Integer, ByVal axis As Integer, Motion As MCMotion) As Integer  
**MCCL command:** SA, DS, SV, MV, DI, SG, SQ, DB, DT, SF, SH, FC, HC, LF, LM, LN, FN, FF, HS, LS, MS

**LabVIEW VI:**



### MCMotion Data Structure

```
typedef struct {

    double    Acceleration;    // Acceleration rate for motion
    double    Deceleration;    // Deceleration rate for motion
    double    Velocity;        // Maximum velocity for motion
    double    MinVelocity;     // Stepper motor jog minimum velocity
    short int Direction;       // Sets velocity mode direction of travel
    double    Gain;            // Proportional gain value for motion
    double    Torque;          // Sets the maximum output torque for servos.
                                // Default output units are volts.
    double    Dead band;       // Sets the position dead band value
    double    DeadbandDelay;   // Time limit axis must remain within dead band
    short int StepSize;        // Sets step size output for stepper motor
    short int Current;         // Full or reduced current stepper motor.
    WORD      HardLimitMode;   // Enables hard (physical) limit switches
    WORD      SoftLimitMode;   // Enables soft (software) limit switches
    double    SoftLimitLow;    // Sets "position" of low soft limit
    double    SoftLimitHigh;   // Sets "position" of high soft limit
    short int EnableAmpFault;  // Controls servo amplifier fault input
    short int Rate;            // Servo - set the feedback loop rate
                                // Stepper - sets max. pulse rate range

} MCMOTION;
```



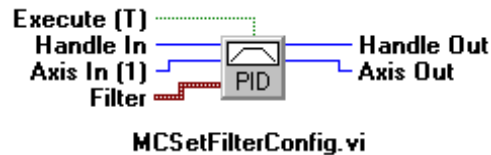
## Set Filter Configuration

This function provides a way of setting all PID filter parameters for a given *wAxis* with a single function call using an initialized **MCFILTER** structure. When you need to setup many parameters for an *wAxis* it is easier to call **MCGetFilter()**, update the **MCFILTER** structure, and write the changes back using **MCSetFilterConfig()**, rather than using a Get/Set function call for each parameter. Note that some less often used parameters will only be accessible from this function and from **MCGetFilterConfig()** - they do not have individual Get/Set functions.

**compatibility:** MC200, MC210  
**see also:** Set Motion Configuration

**C++ Function:** short int MCSetFilterConfig( HCTRLR hCtrlr, WORD wAxis, MCFILTER far\* lpFilter );  
**Delphi Function:** function MCSetFilterConfig( hCtrlr: HCTRLR; wAxis: Word; var lpFilter: MCFILTER ): SmallInt;  
**VB Function:** Function MCSetFilterConfig( ByVal hCtrlr As Integer, ByVal axis As Integer, Filter As MCFilter ) As Integer  
**MCCL command:** SD, FR, SI, IL, VG, AG, DG, SE

**LabVIEW VI:**



## MCFilter Data Structure

```
typedef struct {

    double    DerivativeGain;    // Gain setting for the derivative term of
                                // the PID loop
    double    DerSamplePeriod;   // Time interval, in seconds, between
                                // derivative samples
    double    IntegralGain;      // Gain setting for the integral term of the
                                // PID loop
    double    IntegrationLimit;   // Limits the power the integral gain can use
                                // to reduce error to zero.
    double    VelocityGain;      // Gain setting for the feed-forward gain of
                                // the PID loop
    double    AccelGain;         // Feed-forward acceleration gain setting
    double    DecelGain;         // Feed-forward deceleration gain setting
    double    FollowingError;     // Maximum position error

} MCFILTER;
```

## Set Contour Configuration

This function provides a way of setting all motion parameters for a multi axis contour motion with a single function call using an initialized **MCCONTOUR** structure. When you need to setup many parameters for an *wAxis*, it is easier to call `MCGetContourConfig( )`, update the **MCCONTOUR** structure, and write the changes back using `MCSetContourConfig( )`, 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 `MCGetContourConfig( )` - they do not have individual Get/Set functions.

**compatibility:** MC200, MC210, MC260

**see also:** Set Motion Configuration

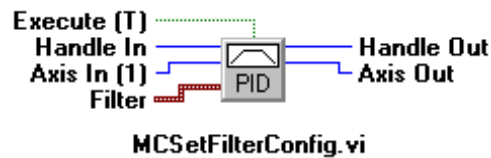
**C++ Function:** `short int MCSetContourConfig( HCTRLR hCtrlr, WORD wAxis, MCCONTOUR far* lpContour );`

**Delphi Function:** `function MCSetContourConfig( hCtrlr: HCTRLR; wAxis: Word; var lpContour: MCCONTOUR ): SmallInt;`

**VB Function:** `Function MCSetContourConfig (ByVal hCtrlr As Integer, ByVal axis As Integer, contour As MCContour) As Integer`

**MCCL command:** VA, VD, VV, VO

**LabVIEW VI:**



## MCCONTOUR Data Structure

```
typedef struct {  
  
    double    VectorAccel;        // Acceleration value for motion along a  
                                // contour path  
    double    VectorDecel;       // Deceleration value for motion along a  
                                // contour path  
    double    VectorVelocity;    // Maximum velocity for motion along a  
                                // contour path  
    double    VelocityOverride;  // Proportional scaling factor for vector  
                                // velocity, may be changed while axes are in  
                                // motion  
  
} MCCONTOUR;
```

## Set Scale

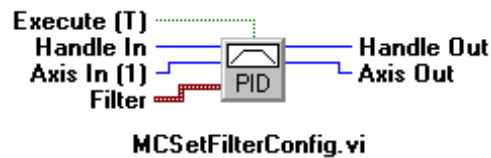
Setting scaling factors allows application programs to talk to the controller in real world units, as opposed to arbitrary "encoder counts". Scaling factors provide a consistent, easy method of relating motion values to the actual physical system being controlled.

This function provides a way of setting all scaling parameters for an axis with a single function call using an initialized **MCSCALE** structure. When you need to setup many parameters for an *wAxis* it is easier to call `MCGetScale( )`, update the **MCSCALE** structure, and write the changes back using `MCSetScale( )`, 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 `MCGetScale( )` - they do not have individual Get/Set functions.

**compatibility:** MC200, MC210, MC260  
**see also:** Set Motion Configuration

**C++ Function:** `short int MCSetScale( HCTRLR hCtrlr, WORD wAxis, MCSCALE far* lpScale );`  
**Delphi Function:** `function MCSetScale( hCtrlr: HCTRLR; wAxis: Word; var lpScale: MCSCALE ): SmallInt;`  
**VB Function:** `Function MCSetScale (ByVal hCtrlr As Integer, ByVal axis As Integer, scal As MCScale) As Integer`  
**MCCL command:** UK, UO, UR, US, UT, UZ

**LabVIEW VI:**



## MCSCALE Data Structure

```
typedef struct {

    double    Constant;           // Scale the analog output of a servo
    double    Offset;             // Offset index position of the encoder
    double    Rate;               // Change the base time unit for motion,
                                // default unit is seconds
    double    Scale;              // Change from encoder counts to 'real
                                // world' units
    double    Zero;               // Set a 'soft' zero position from axis
                                // zero (home)
    double    Time;               // Time factor for Wait functions

} MCSCALE;
```

## Set Jog

This function provides a way of setting all jogging parameters for an axis with a single function call using an initialized **MCJOG** structure. When you need to setup many parameters for an *wAxis* it is easier to call **MCGetJog()**, update the **MCJOG** structure, and write the changes back using **MCSetJog()**, 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 **MCGetJog()** - they do not have individual Get/Set functions.

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

**compatibility:** MC200, MC210, MC260

**see also:** Enable Jog

**C++ Function:** short int MCSetJogConfig( HCTRLR hCtrl, WORD wAxis, MCJOG far\* lpJog );

**Delphi Function:** function MCSetJogConfig( hCtrl: HCTRLR; wAxis: Word; var lpJog: MCJOG ): SmallInt;

**VB Function:** Function MCSetJogConfig (ByVal hCtrl As Integer, ByVal axis As Integer, jog As MCJog) As Integer

**MCCL command:** JA, JM, JD, JG, JO

**LabVIEW VI:** Not supported

## MCJOG Data Structure

```
typedef struct {  
  
    double    Acceleration;           // Jog acceleration rate  
    double    MinVelocity;            // Stepper minimum velocity  
    double    Dead band;              // Voltage threshold below which no  
jog           // motion will occur  
    double    Gain;                   // Defines the jog maximum velocity  
    double    Offset;                 // Specifies 'null' (no motion) of  
                                     // joystick in volts  
  
} MCSCALE;
```

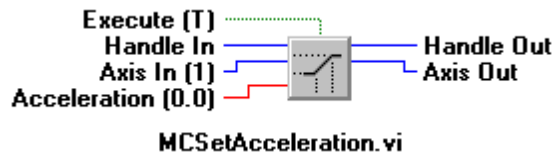
## Set Acceleration

Set the maximum acceleration rate for an axis. The default units for the command parameter are encoder counts (or steps) per second per second.

**compatibility:** MC200, MC210, MC260  
**see also:** Set Deceleration, Set Velocity, Set Motion Config

**C++ Function:** void MCSetAcceleration( HCTRLR hCtrl, WORD wAxis, double Rate );  
**Delphi Function:** procedure MCSetAcceleration( hCtrl: HCTRLR; wAxis: Word; Rate: Double );  
**VB Function:** Sub MCSetAcceleration (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal rte As Double)  
**MCCL command:** SA

**LabVIEW VI:**



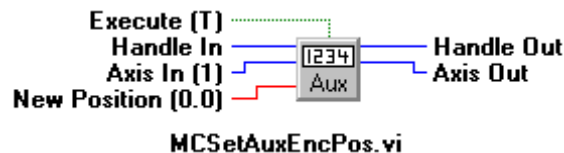
## Set Auxiliary Encoder Position

This command causes axis a auxiliary encoder position to be set to n. This encoder input is available on both the MC200 and MC260 modules, and is used for loop closure when a MC260 is controlling a closed loop stepper. The auxiliary encoder of a MC200 is used for position verification only, it cannot be used for dual loop positioning. For defining the home position of the primary encoder, see the Define Home command.

**compatibility:** MC200, MC210, MC260  
**see also:** Set Position

**C++ Function:** void MCSetAuxEncPos( HCTRLR hCtrl, WORD wAxis, double Position );  
**Delphi Function:** procedure MCSetAuxEncPos( hCtrl: HCTRLR; wAxis: Word; Position: Double );  
**VB Function:** Sub MCSetAuxEncPos (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal position As Double)  
**MCCL command:** AH

**LabVIEW VI:**



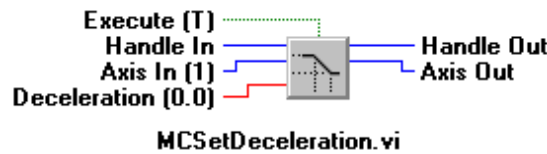
## Set Deceleration

Defines the deceleration rate for an axis. The default units for the command parameter are encoder counts (or steps) per second per second.

**compatibility:** MC200, MC210, MC260  
**see also:** Set Motion Configuration, Set Acceleration, Set Velocity

**C++ Function:** void MCSetDeceleration( HCTRLR hCtrl, WORD wAxis, double Rate );  
**Delphi Function:** procedure MCSetDeceleration( hCtrl: HCTRLR; wAxis: Word; Rate: Double );  
**VB Function:** Sub MCSetDeceleration (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal rte As Double)  
**MCCL command:** DS

**LabVIEW VI:**



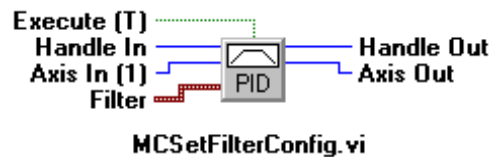
## Set Gain

This function is used to set the proportional gain of a servo's feedback loop. Increasing the proportional gain has the effect of stiffening the force holding a servo in position. The parameter to this command has default units of volts per encoder count. This command should not be used for open loop stepper axes. See the description on **Tuning the Servo** section in the **Motion Control** chapter.

**compatibility:** MC200, MC210  
**see also:** Set Filter Configuration

**C++ Function:** long int MCSetGain( HCTRLR hCtrl, WORD wAxis, double Gain );  
**Delphi Function:** function MCSetGain( hCtrl: HCTRLR; wAxis: Word; Gain: Double ): Longint;  
**VB Function:** Function MCSetGain (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal gain As Double) As Long  
**MCCL command:** SG

**LabVIEW VI:**

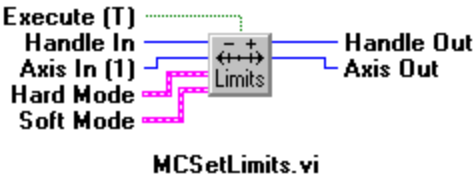


## Set Limits

This function is used to configure and enable both hard and soft limits. 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. If a soft motion limit is enabled, and the respective axis goes 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.

The error flags will remain set until the motor is turned back on with the Enable Axis function. Once the motor is turned back on, it can be moved out of the limit region with any of the standard motion functions. See the description on **Motion Limits** in the **Motion Control** chapter.

**compatibility:** MC200, MC210, MC260  
**see also:** Set Motion Configuration

<b>C++ Function:</b>	long int MCSetLimits( HCTRLR hCtrlr, WORD wAxis, short int HardLimitMode, short int SoftLimitMode, double SoftLimitLow, double SoftLimitHigh );
<b>Delphi Function:</b>	function MCSetLimits( hCtrlr: HCTRLR; wAxis: Word; HardLimMode, SoftLimMode: SmallInt; SoftLimLow, SoftLimHigh: Double );
<b>VB Function:</b>	Function MCSetLimits (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal HardLimitMode As Integer, ByVal SoftLimitMode As Integer, ByVal SoftLimitLow As Double, ByVal SoftLimitHigh As Double) As Long
<b>MCCL command:</b>	HL, LF, LL, LM, LN
<b>LabVIEW VI:</b>	

## Set Module Output Mode

This function is used to set a servo or stepper module's output mode. The available modes are listed in the following tables.

### MC200 Output Mode

Bipolar Analog output, -10V to +10V
Unipolar Analog output, 0V to +10V, direction J3 pin 7
Bipolar PWM signal output on J3 pin 7, 0 - 50% duty cycle
Unipolar PWM signal output on J3 pin 7, 0 – 100% duty cycle, Direction on Analog Output (J3 pin 2)

### MC260 Output Mode

Pulse and Direction outputs (default)
CW and CCW Pulse Outputs

**compatibility:** MC200, MC210, MC260  
**see also:**

<b>C++ Function:</b>	void MCSetModuleOutputMode( HCTRLR hCtrlr, WORD wAxis, WORD wMode );
<b>Delphi Function:</b>	procedure MCSetModuleOutputMode( hCtrlr: HCTRLR; wAxis, wMode: Word );
<b>VB Function:</b>	Sub MCSetModuleOutputMode (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal mode As Integer)
<b>MCCL command:</b>	aOMn a = Axis number n = integer 0, 1, 2, or 3
<b>LabVIEW VI:</b>	Not supported

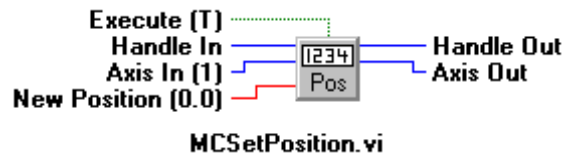
## Set Position

Defines the current position of a motor. From then on, all positions reported for that motor will be relative to that point.

**compatibility:** MC200, MC210, MC260  
**see also:** Get Position

**C++ Function:** void MCSetPosition( HCTRLR hCtrl, WORD wAxis, double Position );  
**Delphi Function:** procedure MCSetPosition( hCtrl: HCTRLR; wAxis: Word; Position: Double );  
**VB Function:** Sub MCSetPosition (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal position As Double)  
**MCCL command:** DH

**LabVIEW VI:**



## Set Operation Mode

This function is used to define the mode of motion of an axis. The available mode are:

MC\_MODE\_CONTOUR selects contouring mode (must also specify wMaster)  
 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.

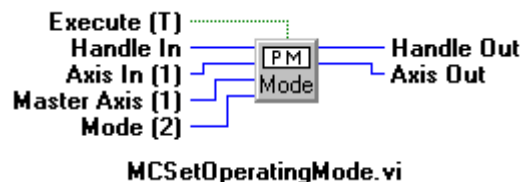
For detailed description of these mode of motion please refer to the **Motion Control chapter** of this user manual.

**compatibility:** MC200, MC210, MC260

**see also:**

**C++ Function:** void MCSetOperatingMode( HCTRLR hCtrl, WORD wAxis, WORD caxis, WORD mode );  
**Delphi Function:** procedure MCSetOperatingMode( hCtrl: HCTRLR; wAxis, wCaxis, wMode: Word );  
**VB Function:** Sub MCSetOperatingMode (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal caxis As Integer, ByVal mode As Integer)  
**MCCL command:** CM, GM, PM, QM, VM

**LabVIEW VI:**



## Set Profile

This function is used to define the type of velocity profile that an axis will use for motion. The supported velocity profiles are:

- Trapezoidal
- S-curve
- Parabolic

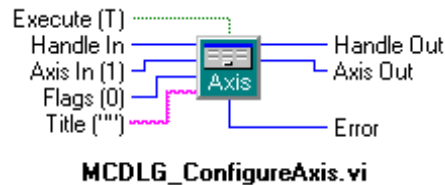
For a description of the performance when using different velocity profiles see the description of **Defining the Characteristics of a Move** in the **Motion Control chapter**.



**compatibility:** MC200, MC210, MC260  
**see also:** PS, PT

**C++ Function:** void MCSetProfile( HCTRLR hCtrl, WORD wAxis, WORD wMode );  
**Delphi Function:** procedure MCSetProfile( hCtrl: HCTRLR; wAxis, wMode: Word );  
**VB Function:** Sub MCSetProfile (ByVal hCtrl As Integer, ByVal axis As Integer, ByVal mode As Integer)  
**MCCL command:** PP, PS, PT

**LabVIEW VI:**



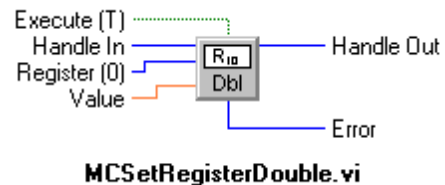
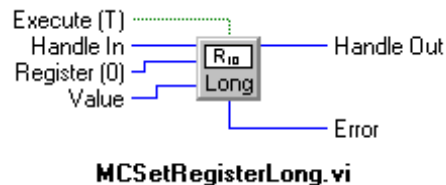
## Set Register

Loads the specified DCX User Register (0 – 255) with a ‘long’ or ‘double’ value. This functional does not support setting the value of a background task ‘private user register’. For more information on background tasks and ‘private user registers’ please refer to the description of **Multi-Tasking** in the **Appendix**.

**compatibility:** N/A  
**see also:** Get Register

**C++ Function:** long int MCSetRegister( HCTRLR hCtrl, long int nRegister, void far\* Value, long int nType );  
**Delphi Function:** function MCSetRegister( hCtrl: HCTRLR; nRegister: Longint; var Value: Pointer; nType: Longint ): Longint;  
**VB Function:** Function MCSetRegister (ByVal hCtrl As Integer, ByVal reg As Long, Value As Any, ByVal argtype As Long) As Long  
**MCCL command:** AL, AR

**LabVIEW VI:**



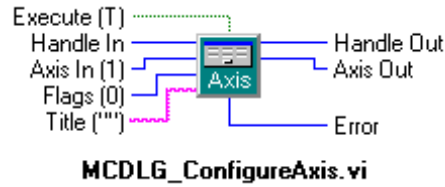
## Set Servo Output Phasing

This function is used to set a servo module's output phasing. The phase of the output will determine whether the module drives the servo in a direction that reduces position error, or increases it.

**compatibility:** MC200, MC210  
**see also:**

**C++ Function:** void MCSetServoOutputPhase( HCTRLR hCtrl, WORD wAxis, WORD wPhase );  
**Delphi Function:** procedure MCSetServoOutputPhase( hCtrl: HCTRLR; wAxis, wPhase: Word );  
**VB Function:** Sub MCSetServoOutputPhase (ByVal hCtrl As Integer, ByVal axis As Integer, ByVal mode As Integer)  
**MCCL command:** PH

**LabVIEW VI:**



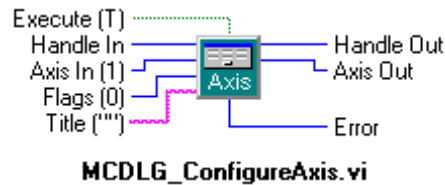
## Set Torque

Sets maximum output level for servos. When an axis is operating in torque mode, this command sets the continuous output level. The default units for the command parameter are volts. See the description of **Torque Mode Output Control** in the **Applications Solutions** chapter.

**compatibility:** MC200, MC210  
**see also:**

**C++ Function:** long int MCSetTorque( HCTRLR hCtrl, WORD wAxis, double Torque );  
**Delphi Function:** function MCSetTorque( hCtrl: HCTRLR; wAxis: Word; Torque: Double ): Longint;  
**VB Function:** Not supported  
**MCCL command:** QM, SQ

**LabVIEW VI:**



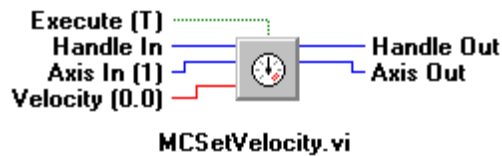
## Set Velocity

Set the maximum velocity for a given axis. The default units for the command parameter are encoder counts (or steps) per second.

**compatibility:** MC200, MC210, MC260  
**see also:** Set Acceleration, Set Deceleration, Set Motion Configuration

**C++ Function:** void MCSetVelocity( HCTRLR hCtrl, WORD wAxis, double Rate );  
**Delphi Function:** procedure MCSetVelocity( hCtrl: HCTRLR; wAxis: Word; Rate: Double );  
**VB Function:** Sub MCSetVelocity (ByVal hCtrl As Integer, ByVal axis As Integer, ByVal rte As Double)  
**MCCL command:** SV

**LabVIEW VI:**



## Set Vector Velocity

This command specifies the maximum velocity for motion along a contour path. It should be issued to the controlling axis prior to the first Contour Path command. When a Contour Path command is issued, the current vector velocity will be stored with the move in the motion table. The Vector Velocity command can also be issued to the controlling axis while motion is in progress, but it won't have any effect on the contour path motions already issued. To adjust the velocity of motions already in progress, use the Velocity Override command. See the description of **Contour Motion (lines and arcs)** in the **Motion Control chapter**.

**compatibility:** MC200, MC210, MC260  
**see also:** Set Contour Configuration

**C++ Function:** short int MCSetContourConfig( HCTRLR hCtrlr, WORD wAxis, MCCONTOUR far\* lpContour );  
**Delphi Function:** function MCSetContourConfig( hCtrlr: HCTRLR; wAxis: Word; var lpContour: MCCONTOUR ): SmallInt;  
**VB Function:** Function MCSetContourConfig (ByVal hCtrlr As Integer, ByVal axis As Integer, contour As MCContour) As Integer  
**MCCL command:** VV  
**LabVIEW VI:** Not supported

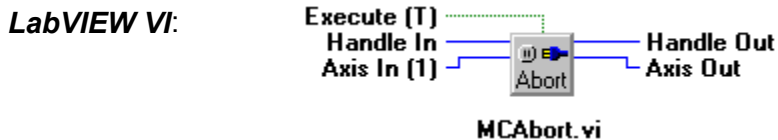
## Motion Functions

### Abort

This function serves as an emergency stop. For a servo, motion stops abruptly but leaves the position feedback loop (PID) and the amplifier enabled. For a stepper motor, the pulses from the module will be disabled immediately. For both servos and stepper motors, the target position of the axis is set equal to the present position. This function can be issued to a specific axis, or can be issued to all axes simultaneously by using the *wAxis* set to **MC\_All\_Axes**.

**compatibility:** MC200, MC210, MC260  
**see also:** Stop

**C++ Function:** void MCAbort( HCTRLR hCtrlr, WORD wAxis );  
**Delphi Function:** procedure MCAbort( hCtrlr: HCTRLR; wAxis: Word );  
**VB Function:** Sub MCAbort (ByVal hCtrlr As Integer, ByVal axis As Integer)  
**MCCL command:** AB



### Arc Center

Specifies the center (absolute or relative) for *wAxis* of an arc for contour path motion. 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. The parameter to this command specifies the center of the arc for the selected axis in user units. See the description of **Contour Motion** in the **Motion Control** chapter.

**compatibility:** MC200, MC210, MC260  
**see also:** Contour Path, Set Operating Mode

**C++ Function:** long int MCArcCenter( HCTRLR hCtrlr, WORD wAxis, short int nType, double Position n );  
**Delphi Function:** function MCArcCenter( hCtrlr: HCTRLR; wAxis: Word; nType: SmallInt; Position: Double ): Longint;  
**VB Function:** Function MCArcCenter (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal arctype As Integer, ByVal position As Double) As Long  
**MCCL command:** CA, CR  
**LabVIEW VI:** Not supported

### Arc Ending Angle

Defines the ending angle (absolute or relative) for *wAxis* of an arc for contour path motion. See the description of **Contour Motion** in the **Motion Control** chapter.

Specifies the center (absolute or relative) for *wAxis* of an arc for contour path motion. 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. The parameter to this command specifies the center of the arc for the selected axis in user units. See the description of **Contour Motion** in the **Motion Control** chapter.

**compatibility:** MC200, MC210, MC260  
**see also:** Contour Path, Set Operating Mode

**C++ Function:** long int MCArcEndAngle( HCTRLR hCtrl, WORD wAxis, short int nType, double Angle );  
**Delphi Function:** function MCArcEndAngle( hCtrl: HCTRLR; wAxis: Word; nType: SmallInt; Anglen: Double ): Longint;  
**VB Function:** Function MCArcEndAngle (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal arctype As Integer, ByVal anglen As Double) As Long  
**MCCL command:** EA, ER  
**LabVIEW VI:** Not supported

## Arc Radius

Specifies the radius for *wAxis* 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. For an arc of less than 180 degrees the parameter *Radius* should be a positive value equal to the radius of the arc. For an arc of greater than 180 degrees the parameter *Radius* should be a negative value equal to the radius of the arc. See the description of **Contour Motion** in the **Motion Control** chapter.

**compatibility:** MC200, MC210, MC260  
**see also:** Contour Path, Set Operating Mode

**C++ Function:** long int MCArcRadius( HCTRLR hCtrl, WORD wAxis, short int nType, double Radius );  
**Delphi Function:** function MCArcRadiusr( hCtrl: HCTRLR; wAxis: Word; nType: SmallInt; Radius: Double ): Longint;  
**VB Function:** Function MCArcRadius(ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal arctype As Integer, ByVal radius As Double) As Long  
**MCCL command:** CA, CR  
**LabVIEW VI:** Not supported

## Capture Data

Record motion data (actual position, optimal position, and DAC output) for an axis. See the description of **Record and display Motion Data** in the **Application Solutions** chapter.

**compatibility:** MC200, MC210  
**see also:**

**C++ Function:** long int MCCaptureData( HCTRLR hCtrl, WORD wAxis, long int IPoints, double Period, double Delay );  
**Delphi Function:** function MCCaptureData( hCtrl: HCTRLR; wAxis: Word; IPoints: Longint; Period, Delay: Double ): Longint;  
**VB Function:** Function MCCaptureData (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal points As Long, ByVal period As Double, ByVal delay As Double) As Long  
**MCCL command:** PR  
**LabVIEW VI:** Not supported

## Contour Distance

This function sets the distance for user defined contour path motions. The parameter specifies 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. See the description of **Contour Motion** in the **Motion Control** chapter.

**compatibility:** MC200, MC210, MC260  
**see also:** Contour Path, Set Operation Mode

**C++ Function:** long int MCContourDistance( HCTRLR hCtrl, WORD wAxis, double Distance );  
**Delphi Function:** function MCContourDistance( hCtrl: HCTRLR; wAxis: Word; Distance: Double ): Longint;  
**VB Function:** Function MCContourDistance (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal distance As Double) As Long  
**MCCL command:** CD  
**LabVIEW VI:** Not supported

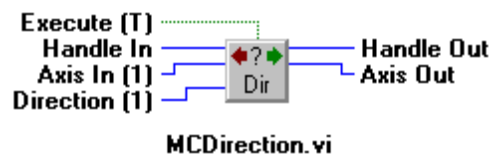
## Direction

Sets the move direction of a motor when in velocity mode.

**compatibility:** MC200, MC210, MC260  
**see also:** GO, enable Velocity mode

**C++ Function:** void MCDirection( HCTRLR hCtrl, WORD wAxis, WORD wDir );  
**Delphi Function:** procedure MCDirection( hCtrl: HCTRLR; wAxis, wDir: Word );  
**VB Function:** Sub MCDirection (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal direc As Integer)  
**MCCL command:** DI

**LabVIEW VI:**



## Enable Axis

Use this function to place one or all servos and stepper 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 or a stepper motor module's Drive Enable output signal will go active. This has the effect of causing servo and stepper motors to hold their 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.

The selected wAxis will be turned on or off depending upon the value of bState. Note that an axis must be enabled before any motion will take place. Issuing this command with wAxis set to MC\_ALL\_AXES will enable or disable all axes installed on hCtrl. This function accepts any non-zero value for bState 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:** MC200, MC210, MC260

**see also:** turn the Motor on

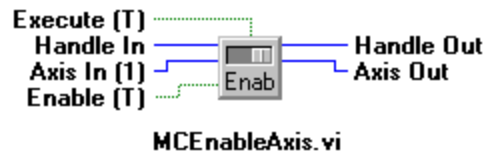
**C++ Function:** void MCEnableAxis( HCTRLR hCtrl, WORD wAxis, short int bState );

**Delphi Function:** procedure MCEnableAxis( hCtrl: HCTRLR; wAxis: Word; bState: SmallInt );

**VB Function:** Sub MCEnableAxis (ByVal hCtrl As Integer, ByVal axis As Integer, ByVal state As Integer)

**MCCL command:** MF, MN

**LabVIEW VI:**



## Enable Backlash

Use this command to set the distance required to nullify the effects of mechanical backlash in the system. The parameter should be equal to one half of the amount the motor must move to take up backlash when it changes direction. The units for this parameter are encoder counts, or the units established by the User Scale command for the axis.

Once the backlash compensation distance is set, enabling backlash compensation will cause the controller to add or subtract the distance from the motor's commanded position during all subsequent moves. If the motor moves in a positive direction, the distance will be added; if the motor moves in a negative direction, it will be subtracted. When the motor finishes a move, it will remain in the compensated position until the next move. See the description on **Backlash Compensation** in the **Application Solutions** chapter of this manual.

**compatibility:** MC200, MC210

**see also:**

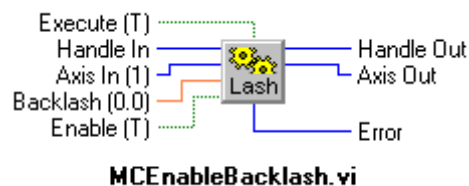
**C++ Function:** long int MCEnableBacklash( HCTRLR hCtrl, WORD wAxis, double Backlash, short int bState );

**Delphi Function:** function MCEnableBacklash( hCtrl: HCTRLR; wAxis: Word; Backlash: Double; bState: SmallInt ): Longint;

**VB Function:** Function MCEnableBacklash (ByVal hCtrl As Integer, ByVal axis As Integer, ByVal backlash As Double, ByVal state As Integer) As Long

**MCCL command:** BD, BN, BF

**LabVIEW VI:**



## Enable Jog

This function enables/disables jogging of servo or stepper axes. See the description of **Jogging** in the **Motion Control** chapter.

**compatibility:** MC200, MC210, MC260  
**see also:** Set Jog Configuration

**C++ Function:** void MCEnableJog( HCTRLR hCtrlr, WORD wAxis, short int bState );  
**Delphi Function:** procedure MCEnableJog( hCtrlr: HCTRLR; wAxis: Word; bState: SmallInt );  
**VB Function:** Sub MCEnableJog (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal state As Integer)  
**MCCL command:** JF, JN  
**LabVIEW VI:** Not supported

## Enable Gearing

This function enables/disables gearing and is used to specify the ratio at which the slave axis will move relative to a change in encoder counts (or steps) of the master axis. When gearing is enabled, the slave axis will begin tracking the master axis with the programmed ratio. The controller makes the position calculations using the optimal positions of the master and slave axes when the Set Master command was issued as the starting point. See the description of **Master/Slave motion** in the **Motion Control** chapter.

**compatibility:** MC200, MC210  
**see also:**

**C++ Function:** void MCEnableGearing( HCTRLR hCtrlr, WORD wAxis, WORD wMaster, double ratio, short int bState );  
**Delphi Function:** procedure MCEnableGearing( hCtrlr: HCTRLR; wAxis, wMaster: Word; Ratio: Double; bState: SmallInt );  
**VB Function:** Sub MCEnableGearing (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal maxis As Integer, ByVal ratio As Double, ByVal state As Integer)  
**MCCL command:** SM, SS  
**LabVIEW VI:** Not supported

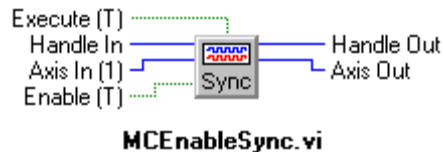
## Enable Synchronization

This function enables or disables synchronized motion for contour path motion for the specified axis. The **MCEnableSync()** 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 **MCGo()** is made. See the description of **Contour Motion** in the **Motion Control** chapter.

**compatibility:** MC200, MC210, MC260  
**see also:** Contour Path

**C++ Function:** void MCEnableSync( HCTRLR hCtrlr, WORD wAxis, short int bState );  
**Delphi Function:** procedure MCEnableSync( hCtrlr: HCTRLR; wAxis: Word; bState: SmallInt );  
**VB Function:** Sub MCEnableSync (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal state As Integer)  
**MCCL command:** NS, NS



**LabVIEW VI:**

## Find the auxiliary encoder index mark

This function is used to initialize an auxiliary encoder at a given position. It will remain in effect until the auxiliary encoder index pulse goes active. At that time the MC\_STAT\_INP\_AUX flag will be set. The current position of the primary axis is acquired (**MCGetPosition**), the axis stopped, moved back to the location of the auxiliary encoder index, and the position of the auxiliary encoder defined using the **MCSetAuxEncPos**.

This function 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 motor's travel (once per revolution in rotary encoders), a typical application will require a coarse home signal to 'qualify' the index pulse. See the description of **Auxiliary Encoders** in the **Application Solutions** chapter.



The function remains pending **until the auxiliary encoder index input of the module goes active**. If the DCX does not 'see' the index input go active the MCAPI will 'lock up'.

**compatibility:**

MC200, MC210, MC260

**see also:**

Get position, Set auxiliary encoder position

**C++ Function:**

```
long int MCFindAuxEncIdx( HCTRLR hCtrlr, WORD wAxis );
```

**Delphi Function:**

```
function MCFindAuxEncIdx( hCtrlr: HCTRLR; wAxis: Word ): Longint;
```

**VB Function:**

```
Function MCFindAuxEncIdx( ByVal hCtrlr As Integer, ByVal axis As Integer ) As Long
```

**MCCL command:**

AF

**LabVIEW VI:**

Not supported

## Find Edge

This function is used to initialize a stepper motor at a given position. At that time the current position of the axis will be set to the value of the Position parameter. This function does not cause any motion to be started or stopped. It is up to the user to initiate motor motion before issuing the function, and to stop any motion after it completes. See the description of **Homing Axes** in the **Motion Control** chapter.



The function remains pending **until the home input of the module goes active**. If the DCX does not 'see' the home input go active the MCAPI will 'lock up'. MCRReset( ) will 'unlock' the MCAPI but user will need to redefine all parameters with **MCDLG\_RestoreAxis( )**.

**compatibility:** MC260  
**see also:** Find Index

**C++ Function:** long int MCFindEdge( HCTRLR hCtrl, WORD wAxis, double Position );  
**Delphi Function:** function MCFindEdge( hCtrl: HCTRLR; wAxis: Word; Position: Double ): Longint;  
**VB Function:** Function MCFindEdge (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal position As Double) As Long  
**MCCL command:** FE  
**LabVIEW VI:** Not supported

## Find the index mark of the encoder (servo)

This function is used to initialize a servo's encoder at a given position. It will remain in effect until the encoder index pulse goes active. At that time the current position of the servo will be defined to be = *Position*. This function 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. See the description of **Homing Axes** in the **Motion Control** chapter.



The function remains pending **until the encoder index input of the module goes active**. If the DCX does not 'see' the index input go active the MCAPI will 'lock up'.

**compatibility:** MC200, MC210  
**see also:** Wait for Edge

**C++ Function:** long int MCFindIndex( HCTRLR hCtrl, WORD wAxis, double Position );  
**Delphi Function:** function MCFindIndex( hCtrl: HCTRLR; wAxis: Word; Position: Double ): Longint;  
**VB Function:** Function MCFindIndex (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal position As Double) As Long  
**MCCL command:** FI  
**LabVIEW VI:** Not supported

## Go

Causes one or all axes to begin motion in velocity or contour mode. In contour mode, synchronization must be enabled.

**compatibility:** MC200, MC210, MC260  
**see also:** Set Operation Mode

**C++ Function:** void MCGo( HCTRLR hCtrl, WORD wAxis );  
**Delphi Function:** procedure MCGo( hCtrl: HCTRLR; wAxis: Word );  
**VB Function:** Sub MCGo (ByVal hCtrlr As Integer, ByVal axis As Integer)  
**MCCL command:** GO

**LabVIEW VI:**



## GoEx

Causes one or all axes to begin motion in velocity or contour mode. To enable cubic splining while in contour mode set the value of Param to 1.0.

**compatibility:** MC200, MC210, MC260

**see also:** Set Operation Mode

**C++ Function:** long int MCGoEx( HCTRLR hCtrlr, WORD wAxis, double Param );

**Delphi Function:** function MCGoEx( hCtrlr: HCTRLR; wAxis: Word; Param: Double ): Longint;

**VB Function:** Function MCGoEx (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal Param As Double) As Long

**MCCL command:** GO, SN

**LabVIEW VI:** Not supported

## Go Home

Causes the specified axis to move to the position last defined as home. This is equivalent to a Move Absolute command, where the destination is 0.0 or the offset of the home position.

**compatibility:** MC200, MC210, MC260

**see also:** Move absolute, Find Edge, Find Index

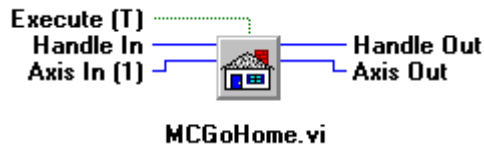
**C++ Function:** void MCGoHome( HCTRLR hCtrlr, WORD wAxis );

**Delphi Function:** procedure MCGoHome( hCtrlr: HCTRLR; wAxis: Word );

**VB Function:** Sub MCGoHome (ByVal hCtrlr As Integer, ByVal axis As Integer)

**MCCL command:** GH

**LabVIEW VI:**



## arm the Index mark capture (servo)

This function is used to arm the capturing of the index mark of a servo. After this function has been called, the **MC\_STAT\_INP\_INDEX** flag will be true when the encoder index mark is captured.

**compatibility:** MC200, MC210

**see also:** Wait for Index

<b>C++ Function:</b>	long int MCIndexArm( HCTRLR hCtrl, WORD wAxis, double Position );
<b>Delphi Function:</b>	function MCIndexArm( hCtrl: HCTRLR; wAxis: Word; Position: Double ): Longint;
<b>VB Function:</b>	Function MCIndexArm (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal position As Double) As Long
<b>MCCL command:</b>	IA, WI
<b>LabVIEW VI:</b>	Not supported

## Learn Point

Used for storing the current position or target position of one or all axes in the DCX's point memory. Points stored in the point memory can be used by the **Move Point** function to repeat a stored motion pattern. See the description of **Learning/ Teaching Points** in the **Application Solutions** chapter.

**compatibility:** MC200, MC210, MC260  
**see also:** Learn the target, Move to point

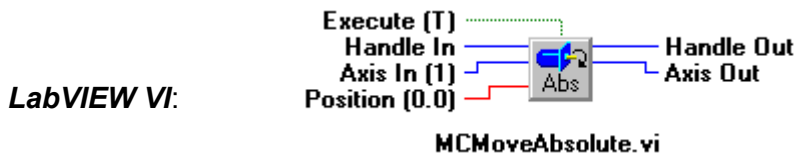
<b>C++ Function:</b>	long int MCLearnPoint( HCTRLR hCtrl, WORD wAxis, long int lIndex, WORD wMode );
<b>Delphi Function:</b>	function MCLearnPoint( hCtrl: HCTRLR; wAxis: Word; lIndex: Longint; wMode: Word ): Longint;
<b>VB Function:</b>	Function MCLearnPoint (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal index As Long, ByVal mode As Integer) As Long
<b>MCCL command:</b>	LP
<b>LabVIEW VI:</b>	Not supported

## Move Absolute

This function executes a motion to an absolute position. A motor number must be specified and that motor must be enabled. 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.

**compatibility:** MC200, MC210, MC260  
**see also:** Enable Axis, Move Relative, Set Operating Mode

<b>C++ Function:</b>	void MCMoveAbsolute( HCTRLR hCtrl, WORD wAxis, double Position );
<b>Delphi Function:</b>	procedure MCMoveAbsolute( hCtrl: HCTRLR; wAxis: Word; Position: Double );
<b>VB Function:</b>	Sub MCMoveAbsolute (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal position As Double)
<b>MCCL command:</b>	MA



## Move to stored point

Used for moving one or more axes to a previously stored point. The parameter *wIndex* 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 parameter *wIndex* MC\_ALL\_AXES all axes will move to the positions stored in the point memory for that point. A specific axis is moved by setting the parameter *wIndex* = the desired axis number. Points are stored in point memory with the Learn Point function. See the description of **Learning/ Teaching Points** in the **Application Solutions** chapter.

**compatibility:** MC200, MC210, MC260  
**see also:** Learn Point

**C++ Function:** long int MCMoveToPoint( HCTRLR hCtrl, WORD wAxis, long int lIndex );  
**Delphi Function:** function MCMoveToPoint( hCtrl: HCTRLR; wAxis: Word; lIndex: Longint ): Longint;  
**VB Function:** Function MCMoveToPoint (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal index As Long) As Long  
**MCCL command:** aMPn a = Axis number n = integer >= 0, <=1536  
**LabVIEW VI:** Not supported

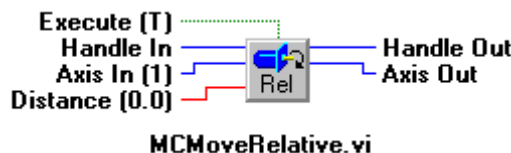
## Move to relative position

This function executes a motion of a relative distance from the current position. A motor number must be specified and that motor must be enabled. 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.

**compatibility:** MC200, MC210, MC260  
**see also:** Enable Axis, Move Absolute, Set Operating Mode

**C++ Function:** void MCMoveRelative( HCTRLR hCtrl, WORD wAxis, double Distance );  
**Delphi Function:** procedure MCMoveRelative( hCtrl: HCTRLR; wAxis: Word; Distance: Double );  
**VB Function:** Sub MCMoveRelative (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal distance As Double)  
**MCCL command:** MR

**LabVIEW VI:**



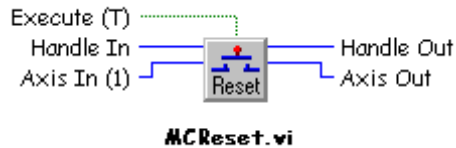
## DCX Reset

The *wIndex* parameter specifies a reset of the entire controller or a specific axis. When executed, all default conditions such as acceleration and velocity will be restored and the axes will be placed in the "off" state.

**compatibility:** MC200, MC210, MC260, MC400, MC5X0  
**see also:** Default Settings in the Appendix

**C++ Function:** void MCRReset( HCTRLR hCtrl, WORD wAxis );  
**Delphi Function:** procedure MCRReset( hCtrl: HCTRLR; wAxis: Word );  
**VB Function:** Sub MCRReset (ByVal hCtrlr As Integer, ByVal axis As Integer)  
**MCCL command:** aRT a = Axis number

**LabVIEW VI:**



## Set Jog

Enables or disables jogging for the axis specified by wAxis. The selected wAxis should be configured for jogging using the **MCSetJogConfig()** function before being enabled by this function.

**compatibility:** MC200, MC210, MC260

**see also:** Get Jog, Set Jog

**C++ Function:** short int MCSetJogConfig( HCTRLR hCtrl, WORD wAxis, MCJOG far\* lpJog );

**Delphi Function:** function MCSetJogConfig( hCtrl: HCTRLR; wAxis: Word; var lpJog: MCJOG ): SmallInt;

**VB Function:** Function MCSetJogConfig (ByVal hCtrlr As Integer, ByVal axis As Integer, jog As MCJog) As Integer

**MCCL command:** JF, JN

**LabVIEW VI:** Not supported

## Stop

Used to stop one or all motors. It differs from the Abort function in that motors will decelerate at their preset rate, instead of stopping abruptly. This function can be issued to a specific axis, or can be issued to all axes. See the description of **Continuous Velocity Motion** in the **Motion Control** chapter.

**compatibility:** MC200, MC210, MC260

**see also:** Abort

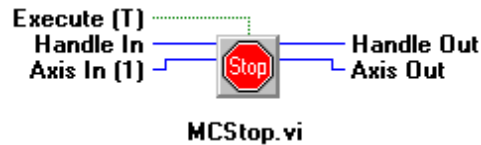
**C++ Function:** void MCStop( HCTRLR hCtrl, WORD wAxis );

**Delphi Function:** procedure MCStop( hCtrl: HCTRLR; wAxis: Word );

**VB Function:** Sub MCStop (ByVal hCtrlr As Integer, ByVal axis As Integer)

**MCCL command:** aST a = Axis number    n = none

**LabVIEW VI:**

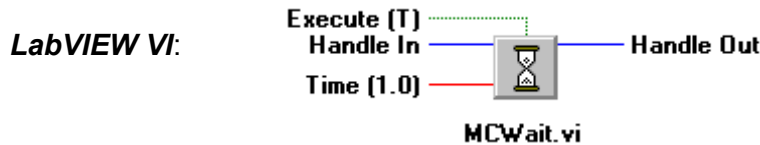


## Wait (a period of time)

Insert a wait period of seconds before going on to the function. If this command was issued from an ASCII interface, it can be aborted by sending an Escape character.

**compatibility:** N/A  
**see also:** Set Scale, Wait for Stop, Wait for Target

**C++ Function:** void MCWait( HCTRLR hCtrl, double Period );  
**Delphi Function:** procedure MCWait( hCtrl: HCTRLR; Period: Double );  
**VB Function:** Sub MCWait (ByVal hCtrlr As Integer, ByVal period As Double)  
**MCCL command:** WA



## Wait for the Coarse Home input (servo and closed loop stepper)

This function is used to initialize a servo or closed loop stepper at a given position. Instruction processing is paused until the coarse home input goes to the specified logic state.

**compatibility:** MC200, MC210, MC260  
**see also:** Find Edge, Find Index, Wait for Index

**C++ Function:** long int MCWaitForEdge( HCTRLR hCtrl, WORD wAxis, short int bState );  
**Delphi Function:** function MCWaitForEdge( hCtrl: HCTRLR; wAxis: Word; nState: SmallInt ): Longint;  
**VB Function:** Function MCWaitForEdge (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal state As Integer) As Long  
**MCCL command:** WE  
**LabVIEW VI:** Not supported

## Wait for Position

This function is used to delay instruction processing until an axis has reached a specific position. The position is specified as a relative distance from the axis zero position. When the specified position has been reached, the DCX will set the breakpoint reached flag in the status word for that axis, and then instruction processing will resume. The axis must be moving before issuing this instruction.



When this function is issued to a DCX-MC260 the position of the auxiliary encoder (**not the step count**) is used for completion of this conditional operation.

**compatibility:** MC200, MC210  
**see also:** Wait for Relative

**C++ Function:** void MCWaitForPosition( HCTRLR hCtrl, WORD wAxis, double Position );  
**Delphi Function:** procedure MCWaitForPosition( hCtrl: HCTRLR; wAxis: Word; Position: Double );  
**VB Function:** Sub MCWaitForPosition (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal position As Double)  
**MCCL command:** IP, WP  
**LabVIEW VI:** Not supported

## Wait for Relative

This function is used to delay instruction processing until an axis has reached a specific position. The position is specified as a relative distance from the current position. When the specified position has been reached, the DCX will set the breakpoint reached flag in the status word for that axis, and then instruction processing will resume. The axis must be moving before issuing this instruction.



When this function is issued to a DCX-MC260 the position of the auxiliary encoder (**not the step count**) is used for completion of this conditional operation.

**compatibility:** MC200, MC210, MC260  
**see also:** Wait for Position

**MCCL command:** WR*n*     *n* = integer or real

**C++ Function:** void MCWaitForRelative( HCTRLR hCtrl, WORD wAxis, double Distance );

**Delphi Function:** procedure MCWaitForRelative( hCtrl: HCTRLR; wAxis: Word; Distance: Double );

**VB Function:** Sub MCWaitForRelative (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal distance As Double)

**LabVIEW VI:** Not supported

## Wait for Stop

Will delay instruction processing until the optimal position (calculated by the DCX trajectory generator) of an axis is equal to the target position of a move. This function can be issued to one or all axes. An optional dwell after the stop may be specified within this command to allow the mechanical system to come to rest.

```
MCMoveAbsolute( hCtrlr, 1, 1000.0 );     // move to position 1000
MCWaitForStop( hCtrlr, 1, 0.005 );     // wait till we're there for 5 msec's
MCMoveAbsolute( hCtrlr, 1, 0.0 );     // move to position 0
```

**comment:** If the **Wait For Stop** function 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.



The function **MCIsStopped** was added to the MCAPI at version 2.20. This provides the same capability as the **MCWaitForStop** without the possibility of locking up the communication of the DCX.



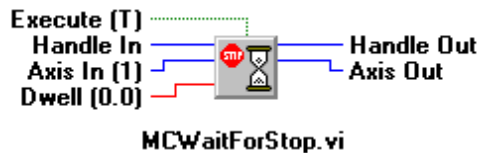


Note: Trajectory Complete is a digital event that occurs when the Optimal Position (calculated by the DCX trajectory generator) equals the Current Position. At this point the Trajectory Complete (bit 3) status bit will be set. Any following error present during the move will cause the Trajectory Complete status bit **to be set before the axis has stopped moving**. The time parameter of the **Wait For Stop** function allows the user to define the time required for the following error to equal 0

**compatibility:** MC200, MC210, MC260  
**see also:** Wait, Wait For Target

**C++ Function:** void MCWaitForStop( HCTRLR hCtrl, WORD wAxis, double Period );  
**Delphi Function:** procedure MCWaitForStop( hCtrl: HCTRLR; wAxis: Word; Period: Double );  
**VB Function:** Sub MCWaitForStop (ByVal hCtrl As Integer, ByVal axis As Integer, ByVal period As Double)  
**MCCL command:** WS

**LabVIEW VI:**



## Wait For Target

For a servo axis to be considered "at target" it must remain within the Dead band region for the DeadbandDelay period. Dead band and DeadbandDelay are specified in the MCMOTION configuration structure. This function delay instruction processing until these conditions have been met. This function can be issued to one or all axes. An optional dwell after the stop may be specified within this command to allow the mechanical system to come to rest.

Note - Wait For Target should not be used for axes in contour mode.

```
MCMoveAbsolute( hCtrlr, 1, 1000.0 );    // move to position 1000
MCWaitForTarget( hCtrlr, 1, 0.0 );      // wait for target
MCMoveAbsolute( hCtrlr, 1, 0.0 );      // move to position 0
```

**comment:** If the **Wait For Target** function 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.

**compatibility:** MC200, MC210, MC260  
**see also:** Wait, Wait For Stop

**C++ Function:** void MCWaitForTarget( HCTRLR hCtrl, WORD wAxis, double Period );  
**Delphi Function:** procedure MCWaitForTarget( hCtrl: HCTRLR; wAxis: Word; Period: Double );  
**VB Function:** Sub MCWaitForTarget (ByVal hCtrl As Integer, ByVal axis As Integer, ByVal period As Double)

**MCCL command:** WT  
**LabVIEW VI:** Not supported

## Reporting Functions

### Get Configuration

This function allows the application to query the driver about installed controller hardware and capabilities with a single function call using an initialized **MCPARAM** structure. Included are the number and type of axes, digital and analog IO channels, scaling, and contouring. Note that some less often used parameters will only be accessible from this function and from MCGetJog( ) - they do not have individual Get/Set functions.

**compatibility:** MC200, MC210, MC260, MC400, MC5X0

**see also:**

**C++ Function:** void MCGetConfiguration( HCTRLR hCtrlr, MCPARAM far\* lpParam );

**Delphi Function:** procedure MCGetConfiguration( hCtrlr: HCTRLR; var lpParam: MCPARAM );

**VB Function:** Sub MCGetConfiguration (ByVal hCtrlr As Integer, param As MCPParam)

**MCCL command:** Not supported

**LabVIEW VI:** Not supported

### MCPARAM Data Structure

```
typedef struct {  
  
    short int ID;                      // ID number assigned during driver setup  
    short int Controller type;         //  
    short int NumberAxes;              // Number of axes  
    short int DigitalIO;               // Number of digital channels  
    short int AnalogInput;             // Number of analog inputs  
    short int AnalogOutput;            // Number of analog outputs  
    short int AxisType(8);              // Type of motor control modules  
    short int CanDoScaling;             //  
    short int CanDoContouring;          //  
    short int CanChangeProfile;         //  
    short int CanChangeRates;           //  
    short int SoftLimits;               //  
    short int MultiTasking;             //  
    short int AmpFault;                 //  
  
} MCSCALE;
```

## Get Motion Configuration

This function provides a way of initializing a **MCMOTION** structure with the current motion parameters for the given wAxis. When you need to setup many of the parameters for an axis it is easier to call MCGetMotionConfig( ), update the MCMOTION structure, and write the changes back using MCSetMotionConfig( ), rather than using a Get/Set function call for each parameter. Note that some less often used parameters will only be accessible from this function and from MCSetMotionConfig - they do not have individual Get/Set functions. You may not set the wAxis parameter to MC\_ALL\_AXES for this command.

**compatibility:** MC200, MC210, MC260  
**see also:** Set Motion Configuration

**C++ Function:** short int MCGetMotionConfig( HCTRLR hCtrlr, WORD wAxis, MCMOTION far\* lpMotion );  
**Delphi Function:** function MCGetMotionConfig( hCtrlr: HCTRLR; wAxis: Word; var lpMotion: MCMOTION ): SmallInt;  
**VB Function:** Function MCGetMotionConfig (ByVal hCtrlr As Integer, ByVal axis As Integer, Motion As MCMotion) As Integer  
**MCCL command:** TA, TD, TG, TI, TL, TQ  
**LabVIEW VI:** See the MC Dialog section

## MCMotion Data Structure

```
typedef struct {  
  
    double    Acceleration;    // Acceleration rate for motion  
    double    Deceleration;    // Deceleration rate for motion  
    double    Velocity;        // Maximum velocity for motion  
    double    MinVelocity;     // Stepper motor jog minimum velocity  
    short int Direction;       // Sets velocity mode direction of travel  
    double    Gain;            // Proportional gain value for motion  
    double    Torque;          // Sets the maximum output torque for servos.  
                                // Default output units are volts.  
    double    Dead band;       // Sets the position dead band value  
    double    DeadbandDelay;   // Time limit axis must remain within dead band  
    short int StepSize;        // Sets step size output for stepper motor  
    short int Current;         // Full or reduced current stepper motor.  
    WORD      HardLimitMode;   // Enables hard (physical) limit switches  
    WORD      SoftLimitMode;   // Enables soft (software) limit switches  
    double    SoftLimitLow;    // Sets "position" of low soft limit  
    double    SoftLimitHigh;   // Sets "position" of high soft limit  
    short int EnableAmpFault;  // Controls servo amplifier fault input  
    short int Rate;           // Servo - set the feedback loop rate  
                                // Stepper - sets max. pulse rate range  
  
} MCMOTION;
```

## Decode Status

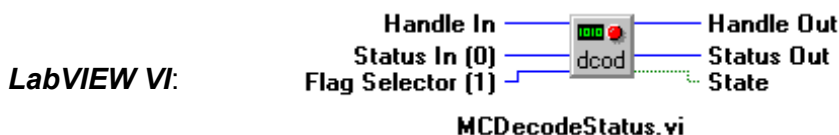
Using this function to test the status word returned by **MCGetStatus( )**. This function returns TRUE if the selected bit is set, or FALSE if the bit is not set or the bit does not apply to this controller type.

**Motor Status Word Constant Lookup Table**

<b>Bit number</b>	<b>DMC_STAT_xxxx Constant</b>
0	MC_STAT_BUSY
1	MC_STAT_MTR_ENABLE
2	MC_STAT_MOD_VEL
3	MC_STAT_TRAJ
4	MC_STAT_DIR
5	MC_STAT_PHASE
6	MC_STAT_HOMED
7	MC_STAT_ERROR
8	MC_STAT_LOOK_INDEX
9	MC_STAT_LOOK_EDGE
10	MC_STAT_FULL_STEP
11	MC_STAT_HALF_STEP
12	MC_STAT_BREAKPOINT
13	MC_STAT_JOGGING
14	MC_STAT_AMP_ENABLE
15	MC_STAT_AMP_FAULT
16	MC_STAT_PLIM_ENAB
17	MC_STAT_PLIM_TRIP
18	MC_STAT_MLIM_ENAB
19	MC_STAT_MLIM_TRIP
20	MC_STAT_PJOG_ENAB
21	MC_STAT_PJOG_ON
22	MC_STAT_MJOG_ENAB
23	MC_STAT_MJOG_ON
24	MC_STAT_INP_INDEX
25	MC_STAT_INP_HOME
26	MC_STAT_INP_AMP
27	none
28	MC_STAT_INP_PLIM
29	MC_STAT_INP_MLIM
30	MC_STAT_INP_PJOG
31	MC_STAT_INP_MJOG

**compatibility:** MC200, MC210, MC260  
**see also:** Get Status

**C++ Function:** long int MCDecodeStatus( HCTRLR hCtrl, DWORD dwStatus, long int lBit );  
**Delphi Function:** function MCDecodeStatus( hCtrl: HCTRLR; dwStatus, dwBit: Longint ): Longint;  
**VB Function:** Function MCDecodeStatus (ByVal hCtrlr As Integer, ByVal status As Long, ByVal bit As Long) As Long  
**MCCL command :** TS



## Error Notify

The **MCErrorNotify()** function registers with the MCAPI a specific window procedure that is to receive message based notification of API errors for this controller handle. 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 and can decide who to have hook the error notification mechanism.

**compatibility:** N/A  
**see also:** Get Error, Translate Error

**C++ Function:** void MCErrNotify( HWND hWnd, HCTRLR hCtrl, DWORD ErrorMask );  
**Delphi Function:** procedure MCErrNotify( hWnd: HWND; hCtrl: HCTRLR; ErrorMask: Longint );  
**VB Function:** Sub MCErrNotify ( ByVal hWnd As Integer, ByVal hCtrl As Integer, ByVal errormask As Long )  
**MCCL command:** Not supported  
**LabVIEW VI:** Not supported

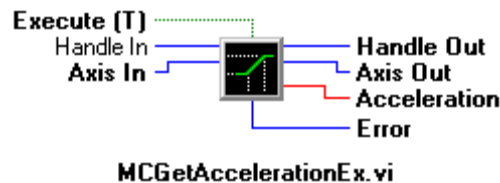
## Get Acceleration

These functions return the current programmed acceleration value for the given axis, in whatever units the axis is configured for. The **MCGetAcceleration()** version returns the acceleration as the function return value, while **MCGetAccelerationEx()** accepts a pointer to a double precision variable for the acceleration result.

**compatibility:** MC200, MC210, MC260  
**see also:** Set Acceleration, Set Motion Configuration

**C++ Function:** long int MCGetAccelerationEx( HCTRLR hCtrl, WORD wAxis, double far\* Acceleration );  
**Delphi Function:** function MCGetAccelerationEx( hCtrl: HCTRLR; wAxis: Word; var Acceleration: Double ): Longint;  
**VB Function:** Function MCGetAccelerationEx ( ByVal hCtrl As Integer, ByVal axis As Integer, accel As Double ) As Long  
**MCCL command:**

**LabVIEW VI:**



## Get Auxiliary Encoder Index

These functions return the position where the auxiliary encoder's index pulse was observed. The **MCGetAuxEnclIdx()** version returns the position as the function return value, while **MCGetAuxEnclIdxEx()** accepts a pointer to a double precision variable for the position result.

**compatibility:** MC200, MC210, MC260  
**see also:**

**C++ Function:** long int MCGetAuxEnclIdxEx( HCTRLR hCtrl, WORD wAxis, double far\* Index );

<b>Delphi Function:</b>	function MCGetAuxEncIdxEx( hCtrl: HCTRLR; wAxis: Word; var Index: Double ): Longint;
<b>VB Function:</b>	Function MCGetAuxEncIdxEx (ByVal hCtrlr As Integer, ByVal axis As Integer, index As Double) As Long
<b>MCCL command:</b>	AZ
<b>LabVIEW VI:</b>	Not supported

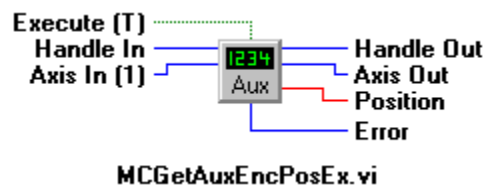
## Get Auxiliary Encoder Position

These functions return the current position of the auxiliary encoder. The **MCGetAuxEncPos( )** version returns the position as the function return value, while **MCGetAuxEncPosEx( )** accepts a pointer to a double precision variable for the position result.

**compatibility:** MC200, MC210, MC260  
**see also:** Set Auxiliary Encoder Position

**C++ Function:** long int MCGetAuxEncPosEx( HCTRLR hCtrl, WORD wAxis, double far\* Position );  
**Delphi Function:** function MCGetAuxEncPosEx( hCtrl: HCTRLR; wAxis: Word; var Position: Double ): Longint;  
**VB Function:** Function MCGetAuxEncPosEx (ByVal hCtrlr As Integer, ByVal axis As Integer, pos As Double) As Long  
**MCCL command:** AT

**LabVIEW VI:**



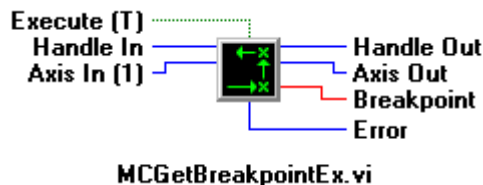
## Get Breakpoint

These functions return the current breakpoint position as placed by the **Wait For Position** or **Wait For Relative** command. The **Get Breakpoint** version returns the breakpoint as the function return value, while **Get Breakpoint Ex** accepts a pointer to a double precision variable for the breakpoint result.

**compatibility:** MC200, MC210, MC260  
**see also:** Wait For Position, Wait For Relative

**C++ Function:** long int MCGetBreakpointEx( HCTRLR hCtrl, WORD wAxis, double far\* Breakpoint );  
**Delphi Function:** function MCGetBreakpointEx( hCtrl: HCTRLR; wAxis: Word; var Breakpoint: Double ): Longint;  
**VB Function:** Function MCGetBreakpointEx (ByVal hCtrlr As Integer, ByVal axis As Integer, bpoint As Double) As Long  
**MCCL command:** TB

**LabVIEW VI:**



## Get Captured Data

This command is used to retrieve the data collected by the most recent **MCaptureData** function call. The three types of position data include; actual position (MC\_DATA\_ACTUAL), Following error (MC\_DATA\_ERROR), and optimal position (MC\_DATA\_OPTIMAL). See the description of **Record and display Motion Data** in the **Application Solutions** chapter.

**compatibility:** MC200, MC210, MC260  
**see also:** Capture Data

**C++ Function:** long int MCGetCaptureData( HCTRLR hCtrl, WORD wAxis, long int IType, long int IStart, long int IPoints, double far\* lpData );

**Delphi Function:** function MCGetCaptureData( hCtrl: HCTRLR; wAxis: Word; IType, IStart, IPoints: Longint; var lpData: Double ): Longint;

**VB Function:** Function MCGetCaptureData (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal mode, ByVal spoint As Long, ByVal npoints As Long, dbuffer As Double) As Long

**MCCL command:** DR

**LabVIEW VI:** Not supported

## Get Contour Configuration

Obtains the contouring configuration for the specified axis. This function allows the application to query an axis about contouring configuration data. You may not set the wAxis parameter to MC\_ALL\_AXES for this command.

**compatibility:** MC200, MC210, MC260  
**see also:** Set Contour Configuration, Set Operating Mode

**C++ Function:** short int MCGetContourConfig( HCTRLR hCtrl, WORD wAxis, MCCONTOUR far\* lpContour );

**Delphi Function:** function MCGetContourConfig( hCtrl: HCTRLR; wAxis: Word; var lpContour: MCCONTOUR ): SmallInt;

**VB Function:** Function MCGetContourConfig (ByVal hCtrlr As Integer, ByVal axis As Integer, contour As MCContour) As Integer

**MCCL command:** N/A

**LabVIEW VI:** Not supported

## Get Contouring Count

Reports the current contour path motion that an axis is performing. The wAxis parameter must be set to the 'controlling' axis of the contour move. The contour count value reported by this function is reset to zero when the **Set Contour Configuration** function is issued with contour mode enabled (MC\_MODE\_CONTOUR).

For each Linear or User Defined Contour Path motion that the controller completes, the contouring count will be incremented by one. For Arc Contour Path motions, the count will be incremented by 2. By counting the number of Contour Path commands that have been issued to the controller (1 for linear, 2 for arc), and comparing it to the response from the TX command, the user can determine on what segment of a continuous path motion the motors are on. The contour count is stored as a 32 bit value (2,147,483,647).



**compatibility:** MC200, MC210, MC260  
**see also:** Set Contour Configuration, Set Operating Mode

**C++ Function:** long int MCGetContouringCount( HCTRLR hCtrl, WORD wAxis );  
**Delphi Function:** function MCGetContouringCount( hCtrl: HCTRLR; wAxis: Word ): Longint;  
**VB Function:** Function MCGetContouringCount (ByVal hCtrlr As Integer, ByVal axis As Integer) As Long  
**MCCL command:** TX  
**LabVIEW VI:** Not supported

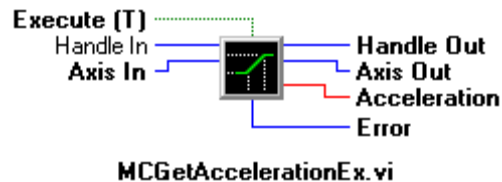
## Get Deceleration

These functions return the current programmed acceleration value for the given axis, in whatever units the axis is configured for. The **MCGetDeceleration( )** version returns the acceleration as the function return value, while **MCGetDecelerationEx( )** accepts a pointer to a double precision variable for the acceleration result.

**compatibility:** MC200, MC210, MC260  
**see also:** Set Deceleration, Set Motion Configuration

**C++ Function:** long int MCGetDecelerationEx( HCTRLR hCtrl, WORD wAxis, double far\* Deceleration );  
**Delphi Function:** function MCGetDecelerationEx( hCtrl: HCTRLR; wAxis: Word; var Deceleration: Double ): Longint;  
**VB Function:** Function MCGetDecelerationEx (ByVal hCtrlr As Integer, ByVal axis As Integer, decel As Double) As Long  
**MCCL command:** DS

**LabVIEW VI:**



## Get Error

Returns the most recent error code for *hCtrl*. The return value is a numeric error code (or MCERR\_NOERROR if there is no error) for the most recent error detected for a specified controller. The error is cleared (set equal to MCERR\_NOERROR) after it has been read. Errors are maintained on a per-handle basis, 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 MCErrNotify( ). This function delivers error messages directly to the window procedure of your choice.

**compatibility:** N/A  
**see also:** Error Notify, Translate Error

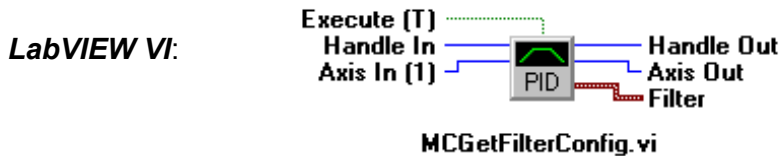
**C++ Function:** short int MCGetError( HCTRLR hCtrl );  
**Delphi Function:** function MCGetError( hCtrl: HCTRLR ): SmallInt;  
**VB Function:** Function MCGetError (ByVal hCtrlr As Integer) As Integer  
**MCCL command:** Not supported  
**LabVIEW VI:** Not supported

## Get Filter Configuration

This function is used to obtain the current PID filter settings for an axis.

**compatibility:** MC200, MC210  
**see also:** set Derivative gain

**C++ Function:** short int MCGetFilterConfig( HCTRLR hCtrl, WORD wAxis, MCFILTER far\* lpFilter );  
**Delphi Function:** function MCGetFilterConfig( hCtrl: HCTRLR; wAxis: Word; var lpFilter: MCFILTER ): SmallInt;  
**VB Function:** Function MCGetFilterConfig (ByVal hCtrlr As Integer, ByVal axis As Integer, Filter As MCFilter) As Integer  
**MCCL command:** TD, TG, TI, TL



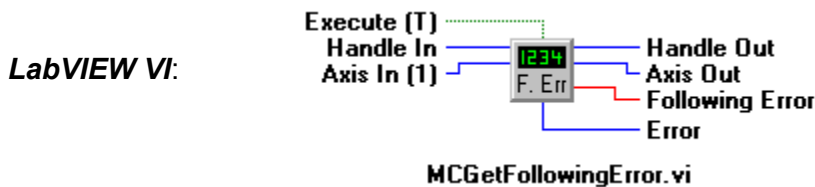
## Get Following Error

Returns the current following error (difference between the actual and the optimal positions) for the specified axis.

Returns the current following error of a servo. This error is the difference between the optimal position (calculated by the trajectory generator) and the current position (decoded encoder pulses).

**compatibility:** MC200, MC210  
**see also:** stop when defined following error exceeded

**C++ Function:** long int MCGetFollowingError( HCTRLR hCtrl, WORD wAxis, double far\* Error );  
**Delphi Function:** function MCGetFollowingError( hCtrl: HCTRLR; wAxis: Word; var Error: Double ): Longint;  
**VB Function:** Function MCGetFollowingError (ByVal hCtrlr As Integer, ByVal axis As Integer, Value As Double) As Long  
**MCCL command:** TF



## Get Gain

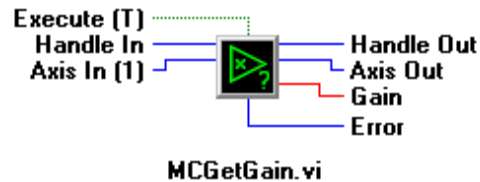
This function returns the current PID filter proportional gain setting of an axis.

**compatibility:** MC200, MC210  
**see also:** Set Gain

**C++ Function:** long int MCGetGain( HCTRLR hCtrl, WORD wAxis, double far\* Gain );  
**Delphi Function:** function MCGetGain( hCtrl: HCTRLR; wAxis: Word; var Gain: Double ): Longint;  
**VB Function:** Function MCGetGain (ByVal hCtrlr As Integer, ByVal axis As Integer, gain As Double) As Long

**MCCL command:** TG

**LabVIEW VI:**



## Get Encoder Index

These functions return the position where the encoder index pulse was observed for the specified axis, in whatever units the axis is configured for. The **MCGetIndex( )** version returns the index position as the function return value, while **MCGetIndexEx( )** accepts a pointer to a double precision variable for the index position result.

The position returned is relative to the encoder's position when the controller was last reset, the axis was homed, or a position was redefined by the Set Position command.

**compatibility:** MC200, MC210

**see also:**

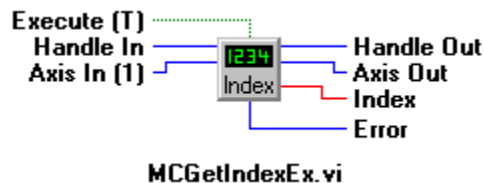
**C++ Function:** long int MCGetIndexEx( HCTRLR hCtrl, WORD wAxis, double far\* Index );

**Delphi Function:** function MCGetIndexEx( hCtrl: HCTRLR; wAxis: Word; var Index: Double ): Longint;

**VB Function:** Function MCGetIndexEx (ByVal hCtrlr As Integer, ByVal axis As Integer, index As Double) As Long

**MCCL command:** TZ

**LabVIEW VI:**



## Get Jog Configuration

Obtains the current jog configuration block for the specified axis. You may not set the wAxis parameter to MC\_ALL\_AXES for this function.

**compatibility:** MC200, MC210, MC260

**see also:**

**C++ Function:** short int MCGetJogConfig( HCTRLR hCtrl, WORD wAxis, MCJOG far\* lpJog );

**Delphi Function:** function MCGetJogConfig( hCtrl: HCTRLR; wAxis: Word; var lpJog: MCJOG ): SmallInt;

**VB Function:** Function MCGetJogConfig (ByVal hCtrlr As Integer, ByVal axis As Integer, jog As MCJog) As Integer

**MCCL command:** Not supported

**LabVIEW VI:** Not supported

## Get Limits

MCGetLimits( ) obtains the current hard and soft limit settings for the specified axis. The limit settings are the same as those reported by the **MCGetMotionConfig( )** function. This function provides a short-hand method for obtaining just the limit settings. You may not set the *wAxis* parameter to MC\_ALL\_AXES for this command.

**compatibility:** MC200, MC210, MC260

**see also:** Get Motion Configuration, Set Limits, Set Motion Configuration

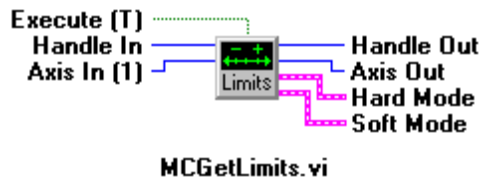
**C++ Function:** long int MCGetLimits( HCTRLR hCtrlr, WORD wAxis, short int far\* HardLimitMode, short int far\* SoftLimitMode, double far\* SoftLimitLow, double far\* SoftLimitHigh );

**Delphi Function:** function MCGetLimits( hCtrlr: HCTRLR; wAxis: Word; var HardLimMode, SoftLimMode: SmallInt; var SoftLimLow, SoftLimHigh: Double ): Longint;

**VB Function:** Function MCGetLimits (ByVal hCtrlr As Integer, ByVal axis As Integer, HardLimitMode As Integer, SoftLimitMode As Integer, SoftLimitLow As Double, SoftLimitHigh As Double) As Long

**MCCL command:** TG

**LabVIEW VI:**



## Get Optimal

The trajectory generator calculates 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 MCFILTER structure. You may not set the *wAxis* parameter to MC\_ALL\_AXES for either of these functions.

**compatibility:** MC200, MC210, MC260

**see also:** Set Acceleration, Set Deceleration, Set Velocity

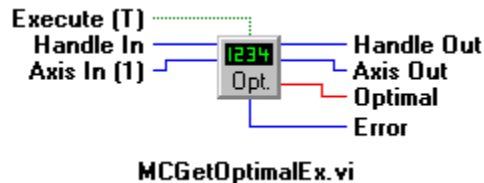
**C++ Function:** long int MCGetOptimalEx( HCTRLR hCtrlr, WORD wAxis, double far\* Optimal );

**Delphi Function:** function MCGetOptimalEx( hCtrlr: HCTRLR; wAxis: Word; var Optimal: Double ): Longint;

**VB Function:** Function MCGetOptimalEx (ByVal hCtrlr As Integer, ByVal axis As Integer, optimal As Double) As Long

**MCCL command:** TO

**LabVIEW VI:**



## Get Position

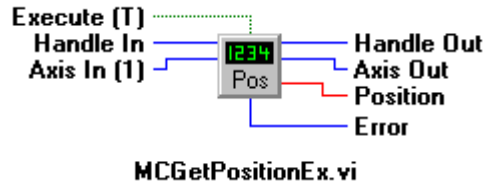
These functions return the current position for the axis selected by *wAxis* specified axis, in whatever units the axis is configured for. The **MCGetPosition( )** version returns the position as the function

return value, while **MCGetPositionEx( )** accepts a pointer to a double precision variable for the position result. You may not set the wAxis parameter to MC\_ALL\_AXES for either of these functions.

**compatibility:** MC200, MC210, MC260  
**see also:** Set Position

**C++ Function:** long int MCGetPositionEx( HCTRLR hCtrl, WORD wAxis, double far\* Position );  
**Delphi Function:** function MCGetPositionEx( hCtrl: HCTRLR; wAxis: Word; var Position: Double ): Longint;  
**VB Function:** Function MCGetPositionEx (ByVal hCtrlr As Integer, ByVal axis As Integer, position As Double) As Long  
**MCCL command:** TP

**LabVIEW VI:**



## Get Profile

This function returns the current acceleration/deceleration profile for the specified axis.

**compatibility:** MC200, MC210, MC260  
**see also:**

**C++ Function:** long int MCGetProfile( HCTRLR hCtrl, WORD wAxis, WORD far\* wProfile );  
**Delphi Function:** function MCGetProfile( hCtrl: HCTRLR; wAxis: Word; var wMode: Word ): Longint;  
**VB Function:** Function MCGetProfile (ByVal hCtrlr As Integer, ByVal axis As Integer, profile As Integer) As Long  
**MCCL command:** Not supported  
**LabVIEW VI:** Not supported

## Get Register

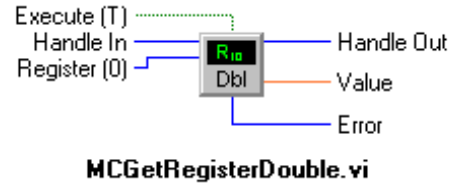
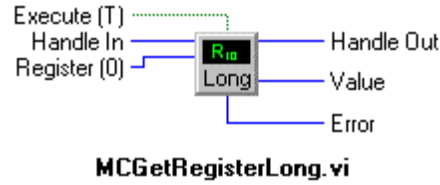
**MCGetRegister( )** and **MCSetRegister( )** allow you to read from and write to, respectively, the general purpose registers on the motion controller. When the command parameter is set to 0 (or not specified), this command reports the contents of User Register zero, which is the accumulator. This command will accept a decimal point and a tenths digit appended to the register number in the command parameter. This digit specifies the number of digits to the right of the decimal point that real number values will be rounded to for display.

When running background tasks on a multitasking controller the only way to communicate with the background tasks is to pass parameters via 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). See the description of **Macros** and **Multi-Tasking** in the **DCX MCCL Commands** chapter of this manual.

**compatibility:** N/A  
**see also:** Set Register

**C++ Function:** long int MCGetRegister( HCTRLR hCtrlr, long int nRegister, void far\* Value, long int nType );  
**Delphi Function:** function MCGetRegister( hCtrlr: HCTRLR; nRegister: Longint; var Value: Pointer; nType: Longint ): Longint;  
**VB Function:** Function MCGetRegister (ByVal hCtrlr As Integer, ByVal reg As Long, Value As Any, ByVal argtype As Long) As Long  
**MCCL command:** TR

**LabVIEW VI:**



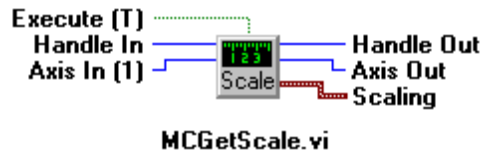
## Get Scale

Obtains the current scaling factors for the specified axis.

**compatibility:** MC200, MC210, MC260  
**see also:** Set Motion Configuration, Set Scale

**C++ Function:** short int MCGetScale( HCTRLR hCtrlr, WORD wAxis, MCSCALE far\* lpScale );  
**Delphi Function:** function MCGetScale( hCtrlr: HCTRLR; wAxis: Word; var lpScale: MCSCALE ): SmallInt;  
**VB Function:** Function MCGetScale (ByVal hCtrlr As Integer, ByVal axis As Integer, scal As MCScale) As Integer  
**MCCL command:** UK, UO, UR, US, UT, UZ

**LabVIEW VI:**



## Get Servo Output Phase

This function returns the current servo output phasing for the specified axis. You may not set the wAxis parameter to MC\_ALL\_AXES for this command.

**compatibility:** MC200, MC210  
**see also:** Set Register

**C++ Function:** long int MCGetServoOutputPhase( HCTRLR hCtrlr, WORD wAxis, WORD far\* wPhase );  
**Delphi Function:** function MCGetServoOutputPhase( hCtrlr: HCTRLR; wAxis: Word; var wPhase: Word ): Longint;  
**VB Function:** Function MCGetServoOutputPhase (ByVal hCtrlr As Integer, ByVal axis As Integer, phase As Integer) As Long  
**MCCL command:** Not supported  
**LabVIEW VI:** Not supported

## Get Status

Returns the 32 bit status word for the selected axis. The MCAPI function **MCDecodeStatus()** provides a way to test the state of an individual status flag. You may not set the wAxis parameter to MC\_ALL\_AXES for this command.

### DCX-MC200, DCX-MC210, and DCX-MC260 Motor status

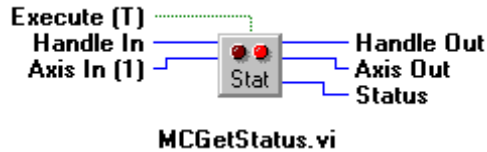
Bit number	Description
0	Busy (motor data being updated)
1	Motor On
2	At Target
3	Trajectory Complete (Optimal = Target)
4	Direction (0 = positive, 1 = negative)
5	Motor Jogging is Enabled
6	Motor homed
7	Motor Error (Limit +/- tripped, max. following error exceeded)
8	Looking For Index (FI, WI)
9	Looking For Edge (FE, WE)
10	Index found
11	Unused
12	Breakpoint Reached (IP, IR, WP, WR)
13	Exceeded Max. Following Error *
14	Amplifier Fault Enabled *
15	Amplifier 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 (MC200, MC210)/Stepper Home (MC260)
25	Coarse home (current state)
26	Amplifier Fault *
27	Auxiliary Encoder Index
28	Limit Positive Input Active (current state)
29	Limit Negative Input Active
30	User Input 1 *
31	User Input 2 *

\* not valid for stepper modules

**compatibility:** MC200, MC210, MC260  
**see also:** Decode Status

**C++ Function:** DWORD MCGetStatus( HCTRLR hCtrl, WORD wAxis );  
**Delphi Function:** function MCGetStatus( hCtrl: HCTRLR; wAxis: Word ): Longint;  
**VB Function:** Function MCGetStatus (ByVal hCtrlr As Integer, ByVal axis As Integer) As Long  
**MCCL command:** TS

LabVIEW VI:



## Get Target position

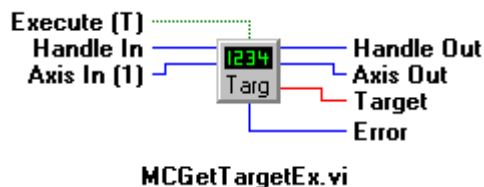
The API move functions **MCMoveAbsolute()** and **MCMoveRelative()** update the target position for an axis. The controller will then generate an optimal trajectory to the target position, and the PID loop will seek to minimize the error (difference between actual and optimal trajectories).

The return value is the target position of the axis selected by *wAxis* for **MCGetTarget()**. For the **MCGetTargetEx()** version 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. You may not set the *wAxis* parameter to **MC\_ALL\_AXES** for either of these functions.

**compatibility:** MC200, MC210, MC260  
**see also:** Move Absolute, Move Relative

**C++ Function:** long int MCGetTargetEx( HCTRLR hCtrlr, WORD wAxis, double far\* Target );  
**Delphi Function:** function MCGetTargetEx( hCtrlr: HCTRLR; wAxis: Word; var Target: Double ): Longint;  
**VB Function:** Function MCGetTargetEx (ByVal hCtrlr As Integer, ByVal axis As Integer, target As Double) As Long  
**MCCL command:** aTTp a = Axis number p = 0, .1, .2, .3, .4, .5

LabVIEW VI:



## Get Torque

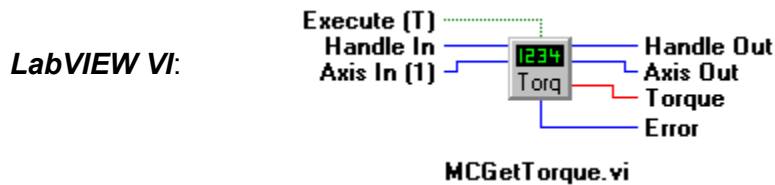
Reports the current DAC output of a servo module. See the description of the **Torque Mode Output Control** in the **Application Solutions** chapter.

Places the current output setting of a servo in the variable specified by the pointer *pTorque*. You may not set the *wAxis* parameter to **MC\_ALL\_AXES** for this command.

**compatibility:** MC200, MC210  
**see also:** Set Torque, Set Operating Mode, Set Motion Configuration

**MCCL command:** TQ  
**C++ Function:** long int MCGetTorque( HCTRLR hCtrlr, WORD wAxis, double far\* Torque );  
**Delphi Function:** function MCGetTorque( hCtrlr: HCTRLR; wAxis: Word; var Torque: Double ): Longint;  
**VB Function:** Function MCGetTorque (ByVal hCtrlr As Integer, ByVal axis As Integer, torque As Double) As Long





## Get Vector Velocity

This function returns the current programmed vector velocity for the specified axis, in whatever units the axis is configured for. The vector velocity value for a particular wAxis 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 wAxis parameter to MC\_ALL\_AXES for this command.

**compatibility:** MC200, MC210  
**see also:** Set Vector Velocity, Set Contour Configuration, Get Contour Configuration

**C++ Function:** long int MCGetTorque( HCTRLR hCtrlr, WORD wAxis, double far\* Torque );  
**Delphi Function:** function MCGetTorque( hCtrlr: HCTRLR; wAxis: Word; var Torque: Double ): Longint;  
**VB Function:** Function MCGetTorque (ByVal hCtrlr As Integer, ByVal axis As Integer, torque As Double) As Long  
**MCCL command:** TQ  
**LabVIEW VI:** Not supported

## Get Velocity

These functions return the current programmed velocity for the specified axis, in whatever units the axis is configured for. The **MCGetVelocity()** version returns the velocity value as the function return value, while **MCGetVelocityEx()** accepts a pointer to a double precision variable for the velocity value. The programmed velocity value for a particular wAxis may also be obtained using the **MCGetMotionConfig()** function. **MCGetVelocity()** 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.

**compatibility:** MC200, MC210, MC260  
**see also:** Set Velocity, Set Motion Configuration, Get Motion Configuration

**C++ Function:** long int MCGetTorque( HCTRLR hCtrlr, WORD wAxis, double far\* Torque );  
**Delphi Function:** function MCGetTorque( hCtrlr: HCTRLR; wAxis: Word; var Torque: Double ): Longint;  
**VB Function:** Function MCGetTorque (ByVal hCtrlr As Integer, ByVal axis As Integer, torque As Double) As Long  
**MCCL command:** TV  
**LabVIEW VI:** Not supported

## Is the axis At the Target

Returns the value *True* if the axis is at the target (**MC\_STAT\_AT\_TARGET**) and *False* if it is not. **MC\_STAT\_AT\_TARGET** is true if the axis remains within the Dead band region for the time specified by the DeadbandDelay period. Dead band and DeadbandDelay are specified in the **MCMOTION**

configuration structure. If a non zero value is defined for *Timeout* , application program processing is suspended until either:

The axis is at target  
**MCIsStopped** has timed out

If *Timeout* = 0 the function returns the current state (complete = true, not complete = false) of the trajectory for the specified axis. This function is similar to **MCWaitForTarget** except that it can only suspend processing of the application program, it does not affect the operation of the DCX controller. This function can be issued to one or all axes. For additional information please refer to the **Motion Complete Indications** section of the **Motion Control** chapter.

**compatibility:** MC200, MC210, MC260  
**see also:** Is Stopped, Wait for Stop, Wait for Target

**C++ Function:** void MCIsAtTarget( HCTRLR hCtrlr, WORD wAxis, double Timeout );  
**Delphi Function:** procedure MCIsAtTarget( hCtrlr: HCTRLR; wAxis: Word; Timeout: Double );  
**VB Function:** Sub MCIsAtTarget (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal timeout As Double)  
**MCCL command:** N/A  
**LabVIEW VI:** Not supported

## Is the axis Stopped

Returns the value *True* if the calculated motion (**MC\_STAT\_TRAJ**) of an axis is complete. **MC\_STAT\_TRAJ** will be true when the optimal position (calculated by the DCX trajectory generator) of an axis is equal to the target position of a move. If a non zero value is defined for *Timeout* , application program processing is suspended until either:

Trajectory has completed (optimal position = target position)  
**MCIsStopped** has timed out

If *Timeout* = 0 the function returns the current state (complete = true, not complete = false) of the trajectory for the specified axis.

This function is similar to **MCWaitForStop** except that it can only suspend processing of the application program, it does not affect the operation of the DCX controller. This function can be issued to one or all axes. An dwell after trajectory is complete is specified by using the function PC Sleep.



Note: Trajectory Complete is a digital event that occurs when the Optimal Position (calculated by the DCX trajectory generator) equals the Current Position. Any following error present during the move will cause the trajectory to be complete **before the axis has stopped moving**. For additional information please refer to the **Motion Complete Indications** section of the **Motion Control** chapter.

**compatibility:** MC200, MC210, MC260  
**see also:** Is at Target, Wait for Stop, Wait for Target

<b>C++ Function:</b>	void MCIsStopped( HCTRLR hCtrlr, WORD wAxis, double Timeout );
<b>Delphi Function:</b>	procedure MCIsStopped( hCtrlr: HCTRLR; wAxis: Word; Timeout: Double );
<b>VB Function:</b>	Sub MCIsStopped (ByVal hCtrlr As Integer, ByVal axis As Integer, ByVal timeout As Double)
<b>MCCL command:</b>	N/A
<b>LabVIEW VI:</b>	Not supported

## Translate Error

This function returns a far pointer to the ASCII error message corresponding to *nError*. If *nError* does not correspond to a valid error message a NULL pointer is returned. It will work with errors returned from **MCGetError( )** and **MCErrNotify( )** error messages.

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.

If you are using the native **MCTranslateError( )** function from an earlier version of MCAPI you may continue to do so, but it is recommended that you migrate to the new version when possible.

<b>compatibility:</b>	N/A
<b>see also:</b>	Error Notify, Get Error

<b>C++ Function:</b>	long int MCTranslateErrorEx( short int nError, LPSTR szBuffer, long int nLength );
<b>Delphi Function:</b>	function MCTranslateErrorEx( nError: SmallInt; szBuffer: PChar; nLength: Longint ): Longint;
<b>VB Function:</b>	Function MCTranslateErrorEx (ByVal errorcode As Integer, ByVal szBuffer As String, ByVal nLength As Long) As Long
<b>MCCL command:</b>	Not supported
<b>LabVIEW VI:</b>	Not supported

## I/O Functions

### 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 for the first four channels must be supplied to the DCX on connector J3, 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

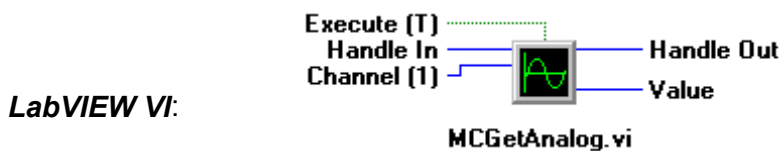
**see also:** Set Analog

**C++ Function:** WORD MCGetAnalog( HCTRLR hCtrlr, WORD wChannel );

**Delphi Function:** function MCGetAnalog( hCtrlr: HCTRLR; wChannel: Word ): Word;

**VB Function:** Function MCGetAnalog (ByVal hCtrlr As Integer, ByVal channel As Integer) As Integer

**MCCL command:** TA



### Configure Digital I/O

Used to configure digital I/O channels as:

Input	MC_DIO_OUTPUT
Output	MC_DIO_OUTPUT
High True	MC_DIO_HIGH
Low True	MC_DIO_LOW

All digital I/O channels on the DCX default to inputs on power-on or reset. The state of a digital I/O channel can be viewed with the **Get Channel** function.

**compatibility:** MC400

**see also:** Get Channel, Enable Channel

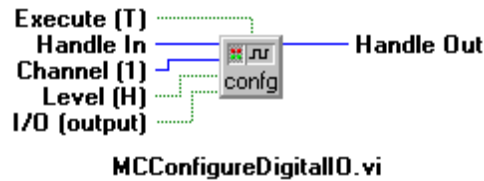
**C++ Function:** short int MCConfigureDigitalIO( HCTRLR hCtrlr, WORD wChannel, WORD wMode );

**Delphi Function:** function MCConfigureDigitalIO( hCtrlr: HCTRLR; wChannel, wMode: Word ): SmallInt;

**VB Function:** Function MCConfigureDigitalIO (ByVal hCtrlr As Integer, ByVal channel As Integer, ByVal mode As Integer) As Integer

**MCCL command:** CH, CI, CL, CT

**LabVIEW VI:**



## 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

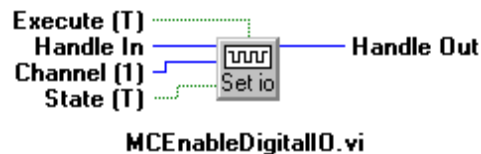
**C++ Function:** void MCEnableDigitalIO( HCTRLR hCtrlr, WORD wChannel, short int bState );

**Delphi Function:** procedure MCEnableDigitalIO( hCtrlr: 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:**

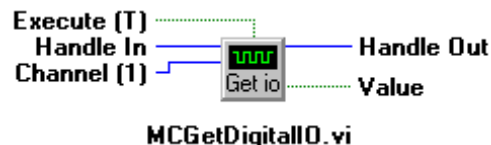
**C++ Function:** short int MCGetDigitalIO( HCTRLR hCtrlr, WORD wChannel );

**Delphi Function:** function MCGetDigitalIO( hCtrlr: HCTRLR; wChannel: Word ): SmallInt;

**VB Function:** Function MCGetDigitalIO (ByVal hCtrlr As Integer, ByVal channel As Integer) As Integer

**MCCL command :** TC

**LabVIEW VI:**



## Get Digital IO Configuration

This function returns the current configuration (in/out/high/low) of the specified digital I/O channel. 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()*.

**compatibility:** MC400  
**see also:** Configure Digital IO

**C++ Function:** long int MCGetDigitalIOConfig( HCTRLR hCtrlr, WORD wChannel, WORD\* wMode );  
**Delphi Function:** function MCGetDigitalIOConfig( hCtrlr: HCTRLR; wChannel: Word; var wMode: Word ): LongInt;  
**VB Function:** Function MCGetDigitalIOConfig (ByVal hCtrlr As Integer, ByVal channel As Integer, mode As Integer) As Long  
**MCCL command :** Not supported  
**LabVIEW VI:** Not supported

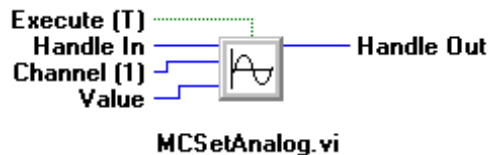
## Set Analog

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

**LabVIEW VI:**



## Wait for Digital IO

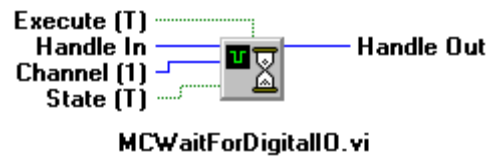
Waits for the specified digital I/O channel to go on or off, depending upon the value of bState.

**compatibility:** MC400  
**see also:** Wait for digital channel on

**C++ Function:** void MCWaitForDigitalIO( HCTRLR hCtrlr, WORD wChannel, short int bState );  
**Delphi Function:** procedure MCWaitForDigitalIO( hCtrlr: 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:**



## Macros and Multi-Tasking

### Cancel Task

Cancels an executing background task. The task should have been previously started with an **MCBlockBegin( )/MCBlockEnd( )** pair. **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( )** and **Macro Commands** in the **Appendix**.

**compatibility:** N/A  
**see also:** Block Begin, Block End

**C++ Function:** long int MCCancelTask( HCTRLR hCtrl, long int ITaskID );  
**Delphi Function:** function MCCancelTask( hCtrl: HCTRLR; ITaskID: Longint ): Longint;  
**VB Function:** Function MCCancelTask (ByVal hCtrlr As Integer, ByVal taskid As Long) As Long  
**MCCL command:** ET  
**LabVIEW VI:** Not supported

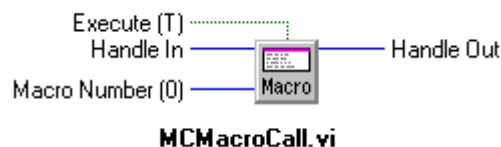
### Macro Call

Macros are normally downloaded using the pmcputs( ) ASCII interface command, using the DCX macro command language (MCCL); or by converting 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. See the description of **Macro Commands** in the **Appendix**.

**compatibility:** N/A  
**see also:** Block Begin, Block End

**C++ Function:** void MCMacroCall( HCTRLR hCtrl, WORD wMacro );  
**Delphi Function:** procedure MCMacroCall( hCtrl: HCTRLR; wMacro: Word );  
**VB Function:** Sub MCMacroCall (ByVal hCtrlr As Integer, ByVal macro As Integer)  
**MCCL command:** MC

**LabVIEW VI:**



### Repeat

Inserts a repeat command into a block command - task, compound command, or macro. This function may only be used within an **MCBlockBegin( )/MCBlockEnd( )** command pair. See the description of **MCBlockBegin( )** and **Macro Commands** in the **Appendix**.

**compatibility:** N/A  
**see also:** Block Begin, Block End



<b>C++ Function:</b>	long int MCRRepeat( HCTRLR hCtrlr, long int ICount );
<b>Delphi Function:</b>	function MCRRepeat( hCtrlr: HCTRLR; ICount: Longint ): Longint;
<b>VB Function:</b>	Function MCRRepeat (ByVal hCtrlr As Integer, ByVal count As Long) As Long
<b>MCCL command:</b>	RP
<b>LabVIEW VI:</b>	Not supported

## MCAPI Driver Functions

### Block Begin

Initiates a block command sequence. Block commands include compound commands, macro definition commands, contour path motions, and multitasking. 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 were 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.

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 *nMode* set to **MC\_BLOCK\_RESETM**. To reset selected blocks of macros, set *nNum* to 1 for RAM-based macros or 2 for Flash memory macros.

All calls to **MCBlockBegin()**, except those with an *nMode* 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 *nMode* 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).

In 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 *lMode* set to **MC\_BLOCK\_CANCEL**.

**compatibility:** N/A  
**see also:** Open

**C++ Function:** long int MCBlockBegin( HCTRLR hCtrlr, long int lMode, long int lNum );

---

<b>Delphi Function:</b>	function MCBlockBegin( hCtrl: HCTRLR; IMode, INum: Longint ): Longint;
<b>VB Function:</b>	Function MCBlockBegin (ByVal hCtrlr As Integer, ByVal mode As Long, ByVal num As Long) As Long
<b>MCCL command:</b>	CP, MD, GT, ET
<b>LabVIEW VI:</b>	Not supported

## Block End

Ends a block command and transmits the compound command, task, macro, or contour path to the controller. 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:** N/A  
**see also:** Open

<b>C++ Function:</b>	long int MCBlockEnd( HCTRLR hCtrl, long int far* ITaskID );
<b>Delphi Function:</b>	function MCBlockEnd( hCtrl: HCTRLR; var ITaskID: Longint ): Longint;
<b>VB Function:</b>	Function MCBlockEnd (ByVal hCtrlr As Integer, TaskID As Long) As Long
<b>MCCL command:</b>	CP, MD, GT, ET
<b>LabVIEW VI:</b>	Not supported

## Close

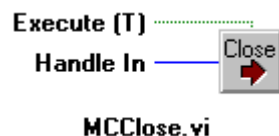
Closes the specified motion controller handle, and is typically called at the end of a program. Following a call to **MCClose( )**, no further calls should be made to the Motion Control API functions with this handle. The exception being **MCOpen( )**, which may be called to open or reopen the API at any time.

By calling **MCClose( )** you notify Windows that you are done with the controller and device driver. After the last user has closed the driver, Windows is 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:** N/A  
**see also:** Open

<b>C++ Function:</b>	short int MCClose( HCTRLR hCtrl );
<b>Delphi Function:</b>	function MCClose( hCtrl: HCTRLR ): SmallInt;
<b>VB Function:</b>	Function MCClose (ByVal hCtrlr As Integer) As Integer
<b>MCCL command:</b>	Not Applicable

**LabVIEW VI:**



## Get Configuration

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:** N/A  
**see also:** Open, Close

**C++ Function:** void MCGetConfiguration( HCTRLR hCtrlr, MCPARAM far\* lpParam );  
**Delphi Function:** procedure MCGetConfiguration( hCtrlr: HCTRLR; var lpParam: MCPARAM );  
**VB Function:** Sub MCGetConfiguration (ByVal hCtrlr As Integer, param As MCParam)  
**MCCL command:** Not supported  
**LabVIEW VI:** Not supported

## Get Version

Returns version information about the MCAPI DLL and, optionally, about the device driver in use for a particular controller. The return version number for the MCAPI DLL and, if hCtrlr is not NULL, the version number for the device driver in use for the controller. If hCtrlr is NULL device driver version info will be zero. 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:** N/A  
**see also:** Open, Close

**C++ Function:** DWORD MCGetVersion( HCTRLR hCtrlr );  
**Delphi Function:** function MCGetVersion( hCtrlr: HCTRLR ): Longint;  
**VB Function:** Function MCGetVersion (ByVal hCtrlr As Integer) As Long  
**MCCL command:** VE  
**LabVIEW VI:** Not supported

## Open

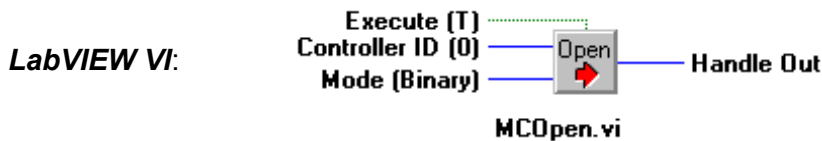
This function returns handle to the specified controller for use in subsequent API calls. The handle will be greater than zero if the open call succeeds, or less than zero if there is an error. Standard error codes (see the file MCERR.H) will be multiplied by -1 to make their values negative and returned in place of a handle if there is an error:

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 MCAPI normally uses the handle to route error messages, and obviously can't do this if the handle is invalid.

**compatibility:** N/A  
**see also:** Close

**C++ Function:** HCTRLR MCOpen( short int nID, WORD wMode, LPCSTR lpszName );  
**Delphi Function:** function MCOpen( hCtrlr: SmallInt; wMode: Word; lpszName: PChar ): HCTRLR;  
**VB Function:** Function MCOpen (ByVal id As Integer, ByVal mode As Integer, ByVal svcname As String) As Integer  
**MCCL command:** Not Applicable



## Reopen

MCREopen( ) may be used to change the mode of an existing handle. The most likely cause for failure is that another open handle exists for the same controller. **MCREopen( )** cannot change a controllers 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.

**compatibility:** N/A  
**see also:** Close, Open

**C++ Function:** long int MCREopen( HCTRLR hCtrlr, WORD wNewMode );  
**Delphi Function:** function MCREopen( hCtrlr: HCTRLR; wNewMode: Word ): Longint;  
**VB Function:** Function MCREopen (ByVal hCtrlr As Integer, ByVal mode As Integer) As Long  
**MCCL command:** Not Applicable  
**LabVIEW VI:** Not Applicable

## Set Timeout

These functions set the timeout period for I/O to a particular controller. The **MCSetTimeout( )** version returns the old timeout setting as the function return value, while **MCSetTimeoutEx( )** accepts a pointer to a double precision variable for the old timeout setting.

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.

**MCSetTimeout( )** is not available in 32-bit versions of the MCAPI (use **MCSetTimeoutEx( )** instead).

**compatibility:** N/A  
**see also:** Close, Open

**C++ Function:** long int MCSetTimeoutEx( HCTRLR hCtrlr, double TimeOut, double far\* OldTimeOut );  
**Delphi Function:** function MCSetTimeoutEx( hCtrlr: HCTRLR; TimeOut: Double; var OldTimeOut: Double ): Longint;  
**VB Function:** Function MCSetTimeoutEx (ByVal hCtrlr As Integer, ByVal timeout As Double, oldtimeout As Double) As Long  
**MCCL command:** Not Applicable  
**LabVIEW VI:** Not Applicable



## Chapter Contents

---

- Motherboard: DCX-AT200
- DCX-MC200 - +/- 10 Volt Analog Servo Motor Control Module
- DCX-MC210 - PWM Motor Drive Servo Control Module
- DCX-MC260 - Stepper Motor Control Module
- DCX-MC400 - 16 channel Digital I/O Module
- DCX-MC5X0 - Analog I/O Module
- DCX-MF300 - RS-232 Communications Interface Module
- DCX-MF310 - IEEE-488 Communications Interface Module



## DCX Specifications

---

### Motherboard: DCX-AT200

Function	6 Axis Motion Controller
Installation	IBM-AT or Compatible or Standalone
Configuration	6 User Installed Modules
Main Processor	Intel i960 RISC
Processor Clock	16 MHz
Code Memory	128k x 16 bit Flash Memory
Data Memory	32k x 16 bit Fast Static RAM 32k x 16 bit Dual Ported Ram (with battery backup option)
Processor Fault Detection	Watchdog Circuit with Reset Relay
Status LED's	Power, Reset, Watch Dog, (6) Motor Error
Standard Communication Interface	ISA Bus (IBM-AT or Compatible) 4 Kilobytes dual ported memory in Memory Address Space Jumper/rotary switch selection of base address
Optional Communication Interfaces	RS-232 Serial Port (Network capable) IEEE-488 Bus
Undedicated Digital I/O Channels	16 TTL (0 – 5 VDC), 1ma max. sink/source
Undedicated Analog Input Channels	4, 8 bit resolution (0 - 5 vdc)
Supply Voltages	+5,+12 and -12 vdc
Form Factor	Full Size PC card (4.3" x 13.6")
Operating Temperature range	0 degrees C to 60 degrees C
Weight	10 oz + 1.2 oz per module (approx.)

## DCX-MC200 - +/- 10 Volt Analog Servo Motor Control Module

Function	Closed Loop Servo Controller with Dual Encoder Inputs
Installation	DCX-AT200 Motion Control Motherboard
Operating Modes	Position, Velocity, Contouring, Torque, Gain, and Joystick
Filter Algorithm	PID with Velocity and Acceleration Feed-Forwards
Filter Update Rate	1, 2 or 4 KHz, software selectable (no integral term for 4 KHz )
Trajectory Generator	Trapezoidal, Parabolic or S-Curve Independent Acceleration and Deceleration
Position Feedback	Incremental Encoder with Index
Position and Velocity Resolution	32 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	1,000,000 Quadrature Counts/Sec.
Encoder Supply Voltage	+5 or +12 vdc, jumper selectable
Axis Inputs	Limit+, Limit-, Coarse Home, Amplifier Fault (TTL compatible, optical isolation available on BF100 interconnect board)
Axis Outputs	Amplifier Enable (TTL compatible)
Jog Control Input	Analog (0 to 5 volts)
General purpose inputs	2 User Inputs, TTL level
Operating Temperature range	0 degrees C to 60 degrees C

## DCX-MC210 - PWM Motor Drive Servo Control Module

Function	Closed Loop Servo Controller with Dual Encoder Inputs
Installation	DCX-AT200 Motion Control Motherboard
Operating Modes	Position, Velocity, Contouring, Torque, Gain, and Joystick
Filter Algorithm	PID with Velocity and Acceleration Feed-Forwards
Filter Update Rate	1, 2 or 4 KHz, software selectable (no integral term for 4 KHz )
Trajectory Generator	Trapezoidal, Parabolic or S-Curve Independent Acceleration and Deceleration
Position Feedback	Incremental Encoder with Index
Position and Velocity Resolution	32 bit
Output	PWM (12 volt @ 1A), 23.4 KHz, 8 bit
Encoder and Index Inputs	Differential or single ended, -7 to +7 vdc max.
Encoder Count Rate	1,000,000 Quadrature Counts/Sec.
Encoder Supply Voltage	+5 or +12 vdc, jumper selectable
Axis Inputs	Limit+, Limit-, Coarse Home, Amplifier Fault (TTL compatible, optical isolation available on BF100 interconnect board)
Axis Outputs	Amplifier Enable (TTL compatible)
Jog Control Input	Analog (0 to 5 volts)
General purpose inputs	2 User Inputs, TTL level
Operating Temperature range	0 degrees C to 60 degrees C

## DCX-MC260 - Stepper Motor Control Module

Function	Open or Closed Loop Stepper Controller
Installation	DCX-AT200 Motion Control Motherboard
Operating Modes	Position, Velocity, Contouring, and Joystick
Trajectory Generator	Trapezoidal, Parabolic or S-Curve Independent Acceleration and Deceleration
Position Feedback	Incremental Encoder with Index (closed loop only)
Position and Velocity Resolution	32 bit
Step Outputs	Pulse/Direction – CW/CCW (software selectable), open collector drivers 50% duty cycle
Step Rates (Software Selectable)	High Speed - 1.0K Steps/Sec. - 1.0M Steps/Sec. Medium Speed - 125 Steps/Sec. - 156K Steps/Sec. Low Speed - 15 Steps/Sec. - 19.5K Steps/Sec.
Aux. Encoder and Index Inputs	Differential or single ended, -7 to +7 vdc max.
Aux. Encoder Count Rate	1,000,000 Quadrature Counts/Sec.
Aux. Encoder Supply Voltage	+5 or +12 vdc, jumper selectable
Axis Inputs	Home, Limit+, Limit-, Null (TTL compatible, optical isolation available on BF160 interconnect board)
Axis Outputs	Driver Enable, Full/Half Step, Full/Half Current, Stopped (TTL compatible)
Jog Control Input	Analog (0 to 5 volts)
General purpose inputs	None
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-AT200 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-AT200 Motion Control Motherboard
Inputs 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
Output Full Scale Adjustment	single turn trim pot
Operating Temperature range	0 degrees C to 60 degrees C

**DCX-MC500 Electrical Specifications**

<b>Parameter</b>	<b>Min.</b>	<b>Max</b>	<b>Unit</b>
Input Resolution	12		Bits
Input Conversion Rate		10	KHz
Input Zero Error			
Using Internal Reference		+/- 3	LSB
Using External Reference		+/- 1/2	LSB
Input Full-Scale Error			
Using Internal Reference		+/- 15	LSB
Using External Reference		+/- 1/2	LSB
Input Zero Temp. Coefficient		0.5	ppm/C
Input Differential Nonlinearity		+/- 1	LSB
Input Total Unadjusted Error			
Using Internal Reference		+/- 15	
Using External Reference		+/- 1	
Input Voltage Range			
Using Internal Reference	0.0	5.0	
Using External Reference	0.0	Vref	
Input Capacitance		8	
Input Leakage Current		100	
External Reference Voltage	4.0	6.0	

<b>Parameter</b>	<b>Min.</b>	<b>Max</b>	<b>Unit</b>
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	5.0	V
	-10.0	+10.0	V

\* These values are for 0 to +5.0 volt outputs

## DCX-MF300 - RS-232 Communications Interface Module

Function	RS-232 Communications Interface module
Installation	DCX-AT200 Motion Control Motherboard
Baud Rates	300 - 19,200
Handshake Protocol	Hardware or XON-XOFF
Operating Temperature range	0 degrees C to 60 degrees C

## DCX-MF310 - IEEE-488 Communications Interface Module

Function	IEEE-488 Communications Interface module
Installation	DCX-AT200 Motion Control Motherboard
Address Selection	DIP switch on module
Data Bus Drivers	Push-Pull or Open Collector
Operating Temperature range	0 degrees C to 60 degrees C

## Chapter Contents

---

- DCX-AT200 Motion Control Motherboard
- DCX-MC200 +/- 10V Servo Motor Control Module
- DCX-MC210 PWM Motor Drive Servo Control Module
- DCX-MC260 Stepper Motor Control Module
- DCX-MC400 Digital I/O Module
- DCX-MC500/MC510/MC520 Analog I/O Module
- DCX-MF300 – RS-232 Interface Module
- DCX-MF310 IEEE-488 Interface Module
- DCX-BF022 Relay Rack Interface
- DCX-BF100 Servo Module Interconnect Board
- DCX-BF160 Stepper Module Interconnect Board



## Connectors, Jumpers, and Schematics

---

### DCX-AT200 Motion Control Motherboard

(Refer to diagram at the end of this appendix)

#### LED Status Indicators

LED #	Color	Description
1	Green	+5V logic supply
2	Yellow	DCX Reset active
3	Yellow	Watchdog circuit tripped
4	Red	Module #1 motor error (following error or limit tripped)
5	Red	Module #2 motor error (following error or limit tripped)
6	Red	Module #3 motor error (following error or limit tripped)
7	Red	Module #4 motor error (following error or limit tripped)
8	Red	Module #5 motor error (following error or limit tripped)
9	Red	Module #6 motor error (following error or limit tripped)

**General Purpose I/O (Digital I/O and Analog inputs) Connector J3**

<b>Pin #</b>	<b>Description</b>
1	+5 VDC
2	ANALOG INPUT #1
3	DIGITAL I/O, CHANNEL 16
4	ANALOG INPUT #3
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 09
12	DIGITAL I/O, CHANNEL 08
13	DIGITAL I/O, CHANNEL 07
14	DIGITAL I/O, CHANNEL 06
15	DIGITAL I/O, CHANNEL 05
16	DIGITAL I/O, CHANNEL 04
17	DIGITAL I/O, CHANNEL 03
18	DIGITAL I/O, CHANNEL 02
19	DIGITAL I/O, CHANNEL 01
20	ANALOG INPUT #4
21	+12 VDC
22	ANALOG INPUT #2
23	ANALOG REF. (input)
24	GROUND
25	-12 VDC
26	GROUND

Mating Connector: 26-pin dual-row IDC female, Circuit Assembly P/N CA-26IDS2-F-SPT or equivalent

**External Reset Switch Connector – J16**

<b>Pin #</b>	<b>Description</b>
1	Reset input (low active)
2	Ground

**Battery Backup Input Connector – J17**

<b>Pin #</b>	<b>Description</b>
1	+5VDC output (from PC power supply) *
2	Battery voltage input (+3.0 VDC nominal)
3	No connect
4	Ground

\* If battery is not used, J17 pins 1 and 2 must be connected together

**DCX-AT200 Configuration Jumpers** – configuration in **bold type** denotes default factory shipping configuration

**JP1 – A/D reference select**

<b>Pins</b>	<b>Description</b>
<b>1 to 2</b>	<b>Select on-board 5.00 volt reference</b>
open	Use external reference (connected to J3 pin 23)

**JP2 – DCX reset source select (one, two or all three)**

<b>Pins</b>	<b>Description</b>
1 to 2	Reserved for factory use
<b>3 to 4</b>	<b>external manual reset from switch connected to connector J16</b>
<b>5 to 6</b>	<b>IBM-PC bus reset</b>

**JP3 – Watchdog circuit enable**

<b>Pins</b>	<b>Description</b>
<b>1 to 2</b>	<b>Enable watchdog circuit</b>
open	disable watchdog circuit

**JP4 – Boot RAM enable**

<b>Pins</b>	<b>Description</b>
1 to 2	Boot code from static memory
<b>open</b>	<b>Boot code from FLASH devices</b>

**JP5 – FPGA Program enable**

<b>Pins</b>	<b>Description</b>
1 to 2	Enable in circuit programming of FPGA
<b>open</b>	<b>Normal DCX operation</b>

**JP6 – PC Interrupts – not supported**

**JP7 – Reserved for factory use**

**JP8 - IBM-PC Interface base memory address select**

<b>Base address</b>	<b>JP8 5 to 6</b>	<b>JP8 3 to 4</b>	<b>JP8 1 to 2</b>
8000 hex	connected	connected	connected
9000 hex	connected	connected	open
A000 hex	connected	open	connected
B000 hex	connected	open	open
C000 hex	open	connected	connected
<b>D000 hex</b>	<b>open</b>	<b>connected</b>	<b>open</b>
E000 hex	open	open	connected
F000 hex	open	open	open

## DCX-AT200 Configuration Jumpers - continued

**JP9 - JP15 – FLASH memory jumper configuration - settings for 1 Meg. FLASH memory devices:**

JP9 1-2  
JP10 1-2  
JP11 2-4  
JP12 1-2  
JP13 1-2  
JP14 open  
JP15 2-3

**JP16 - U19 & U20 Memory type configuration - Factory setting for 32K RAMS:**

JP16 1-2

### DCX Memory Address Rotary Switch Setting

The 16 bit position rotary switch selects where the dual port RAM appears in the host computer's memory map. In the table below, the X digit in the 'Host Address Range', is determined by the setting of jumper JP8. The default factory setting of the switch is 0. If multiple boards are used in a single host, no two boards should have the same switch setting.

Switch Setting	Host Address Range
0	X0000h – X0FFFh
1	X1000h – X1FFFh
2	X2000h – X2FFFh
3	X3000h – X3FFFh
4	X4000h – X4FFFh
5	X5000h – X5FFFh
6	X6000h – X6FFFh
7	X7000h – X7FFFh
8	X8000h – X8FFFh
9	X9000h – X9FFFh
A	XA000h – XAFFFh
B	XB000h – XBFFFh
C	XC000h – XCFFFh
D	XD000h – XDFFFh
E	XE000h – XEFFFh
F	Not a valid address – used to clear macro memory

## IBM-PC Interface Edge Connectors A & B

Component Side	Solder Side
A01 – NC	B01 - GROUND (+5 VOLT RETURN)
A02 - BUS DATA 7 (HIGH TRUE)	B02 - RESET (HIGH TRUE)
A03 - BUS DATA 6	B03 - +5 VDC
A04 - BUS DATA 5	B04 - NC
A05 - BUS DATA 4	B05 - NC
A06 - BUS DATA 3	B06 - NC
A07 - BUS DATA 2	B07 - -12 VDC
A08 - BUS DATA 1	B08 - NC
A09 - BUS DATA 0	B09 - +12 VDC
A10 – IORDY	B10 - GROUND(+,-,12 VOLT RET.)
A11 – NC	B11 - SMEMW (LOW TRUE)
A12 - BUS ADR. 19 (HIGH TRUE)	B12 - SMEMR (LOW TRUE)
A13 - BUS ADR. 18	B13 - NC
A14 - BUS ADR. 17	B14 - NC
A15 - BUS ADR. 16	B15 - NC
A16 - BUS ADR. 15	B16 - NC
A17 - BUS ADR. 14	B17 - NC
A18 - BUS ADR. 13	B18 - NC
A19 - BUS ADR. 12	B19 - NC
A20 - BUS ADR. 11	B20 - NC
A21 - BUS ADR. 10	B21 - NC
A22 - BUS ADR. 09	B22 - NC
A23 - BUS ADR. 08	B23 - IRQ5
A24 - BUS ADR. 07	B24 - NC
A25 - BUS ADR. 06	B25 - NC
A26 - BUS ADR. 05	B26 - NC
A27 - BUS ADR. 04	B27 - NC
A28 - BUS ADR. 03	B28 - BALE (HIGH TRUE)
A29 - BUS ADR. 02	B29 - +5 VDC
A30 - BUS ADR. 01	B30 - NC
A31 - BUS ADR. 00	B31 - GROUND

NC = No Connect

Note: For stand alone applications where the IBM interface is not used:

Connect B11 and B12 to +5V through 4.7K ohm resistor  
Connect B2, A2 - A9 and A11 - A31 to ground

## Auxiliary Connectors

### Asynchronous Serial I/O connector J1

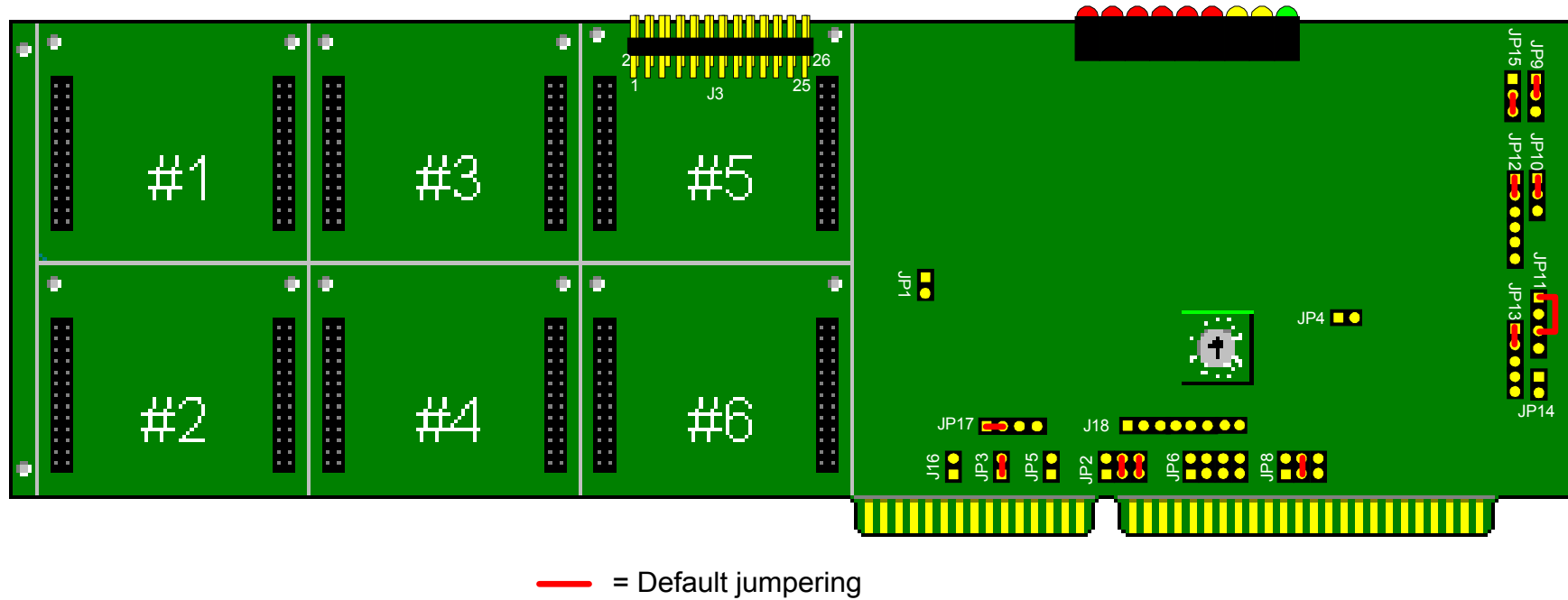
<b>Pin #</b>	<b>Description</b>
1	Ground
2	Serial data input (TTL level)
3	Ground
4	Serial data output (TTL level)
5	Ground
6	Transmit enable (TTL level)
7	+5 VDC
8	+12 VDC
9	-12 VDC
10	NC

Mating Connector: 10-pin dual-row IDC female, Circuit Assembly P/N CA-10IDS2-F-SPT or equivalent

### Auxiliary I/O connector J2

<b>Pin #</b>	<b>Description</b>
1	Reserved for factory use
2	Reserved for factory use
3	Ground
4	Reserved for factory use
5	Ground
6	Reserved for factory use
7	Ground
8	Reserved for factory use
9	+5 VDC
10	Reserved for factory use
11	Counter input
12	Reserved for factory use
13	Watchdog relay contact 1 (normally closed)
14	Watchdog relay contact 2 (normally closed)

Mating Connector: 14-pin dual-row IDC female, Circuit Assembly P/N CA-14IDS2-F-SPT or equivalent



## DCX-MC200 +/- 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. In its default Bipolar output mode, 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. By using the Output Mode command, the output can be changed to Unipolar, where the analog signal range is 0 to +10 volts, and a separate signal is used to indicate the desired direction of velocity or current. The maximum drive current of this signal is +/-10 milliamps.

#### Direction/PWM Output

**connection point:** J3 - pin 7

**signal type:** TTL output

**notes:**

**explanation:**

**Direction** - For servo drives requiring a Unipolar output. The velocity or current command input consists of a magnitude signal and a separate direction signal. The magnitude signal is provided by the modules Analog Command Signal (J3 pin 2) previously described, while this signal provides a digital direction command. The voltage on this output is TTL compatible. This means that it will be between 0 and 0.4 volts (low) to indicate one direction, and between 2.4 and 5.0 volts (high) for the opposite direction. The maximum sink current for this signal when it is low is 4.0 mA, the maximum source current when it is high is 1.0 mA.

**PWM Output** – For servo drives requiring a TTL level PWM command signal. The Analog Command Output (J3 pin 2) is used as the direction signal. The frequency of the PWM is 1.4648 KHz. For a description see the description of **Laser Cutting Application Solutions** chapter.

#### 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 oF command can be used to disable this signal.

#### **Amplifier Enable Output**

**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 Enable signal will be deactivated. This signal can also be deactivated by the Motor oF command.

#### **User Input 1 and User Input 2**

**connection point:** J3 - pin 12 (User Input #1), J3 - pin 13 (User Input #2)

**signal type:** TTL input

**notes:** 4.7K pull up resistor is connected to the +5V logic supply

**explanation:** These module inputs can be connected to any digital logic signals that the DCX needs to monitor. The state of these inputs (high or low) is recorded in the associated bits of the motor status. These signals have no built in control function in the DCX. It is suggested that these inputs be reserved for monitoring signals related to the respective servo axis.

#### **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 oF 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:** The encoder power jumper JP3 sets the 'mid point' for the differential receiver  
**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. By setting jumpers JP1 and JP2 appropriately, the plus signal inputs can be configured for single ended inputs.

#### **Auxiliary Encoder Inputs** (Phase A, 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 can be used for an auxiliary encoder.

#### **Encoder Power Output**

**connection point:** J3 pin 17  
**signal type:** +5 VDC PC power supply output or +12 VDC PC power supply output  
**notes:** The encoder power jumper JP3 selects +5VDC or +12VDC  
**explanation:** This module pin provides a convenient supply voltage connection for the encoders. The jumper JP3 located on the module can be used to connect either the +5 or +12 volt supply to the Encoder Power pin. The setting of this jumper also selects the threshold voltage for the module's single ended phase and index encoder inputs. When JP1 is set for +5 volts, the threshold will be 2.5 volts, for +12 volts, the threshold will be +6 volts. The threshold voltage determines at what voltage the input changes between on and off.

**SUPPLY CONNECTIONS** (+5, +12, -12, GROUND) - These module pins provide access to the DCX supply voltages.

#### **Joystick Input (A/D Channel, 8 bit)**

**connection point:** J4 pin 1  
**signal type:** 0.0 to +5V analog input  
**notes:** used for joystick control  
**explanation:** This input is used to implement manual jogging of the axis. See the description of **Jogging** in the **Motion Control** chapter.

#### **A/D +5 Volt Reference Output**

**connection point:** J4 pin 3  
**signal type:** precision +5V reference voltage  
**notes:** used for joystick control  
**explanation:** This input is used to implement manual jogging of the axis. See the description of **Jogging** in the **Motion Control** chapter.

#### **Analog Ground**

**connection point:** J4 pin 4  
**signal type:** analog ground for A/D conversion  
**notes:** used for joystick control  
**explanation:** This input is used to implement manual jogging of the axis. See the description of **Jogging** in the **Motion Control** chapter.

## DCX-MC200 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	Direction/PWM Output (TTL level)**
8	Primary Encoder Index + (input, active high)
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 Enable (output, active low, TTL level)**
12	User input 1 (input, active low, with 4.7K ohm pull-up to +5V)
13	User input 2 (input, active low, with 4.7K ohm pull-up to +5V)
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	Primary Encoder Phase A+ (input)*
17	Encoder Power (+5VDC or +12VDC, see jumper JP3)
18	Auxiliary Encoder Index - (input, active low)
19	Primary Encoder Phase A- (input)
20	Primary Encoder Phase B- (input)
21	Auxiliary Encoder Phase A
22	Auxiliary Encoder Phase B
23	Primary Encoder Phase B+ (input)*
24	Auxiliary Encoder Index+ (input, active high)
25	Primary 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 CA-26IDS2-F-SPT or equivalent

### J4 connector pin-out (A/D channels for jogging)

Pin #	Description
1	Analog Input #1
2	Analog Input #2
3	+5 volt reference output
4	Analog ground

## DCX-MC200 Module Configuration Jumpers - configuration in **bold type** denotes default factory shipping configuration

### JP1 – Encoder type (single ended or differential)

<b>Pins</b>	<b>Description</b>
1 to 2 to 3	Single ended encoder, A, B, Z (three pin jumper provided)
<b>open</b>	<b>Differential encoder, A+, A-, B+, B-</b>

### JP2 – Encoder Index Active Level Select)

<b>Pins</b>	<b>Description</b>
1 to 2	Single ended Index, Z+ (Active high)
<b>2 to 3</b>	<b>Single ended Index, Z- (active low)</b>
open	Differential Index, Z+ and Z-

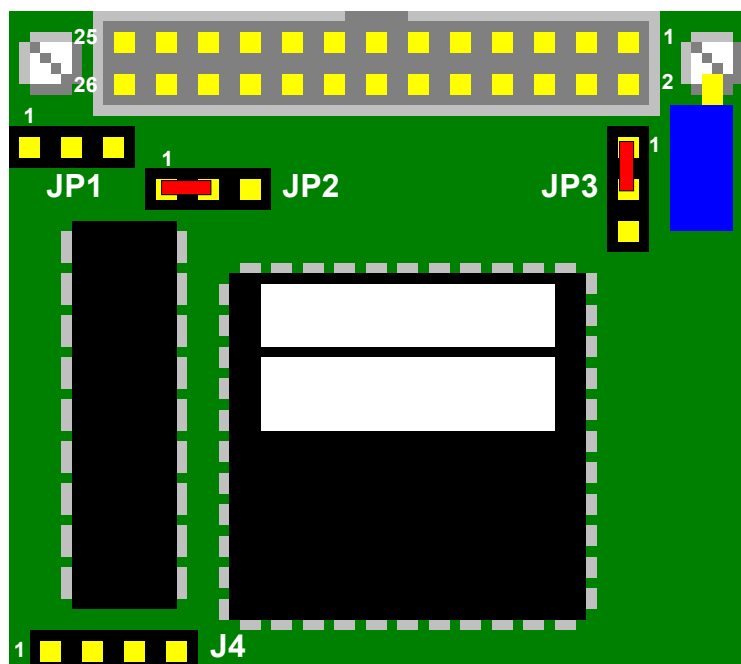
### JP3 – Encoder Power Select (+5VDC or +12 VDC)

<b>Pins</b>	<b>Description</b>
<b>1 to 2</b>	<b>+5 VDC encoder supply on J3 pin 17</b>
2 to 3	+12 VDC encoder supply on J3 pin 17

### DCX-MC200 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-MC200 Module Layout



## DCX-MC210 PWM Motor Drive Servo Control Module

### SIGNAL DESCRIPTIONS:

#### Motor Drive Outputs

**connection point:** J3 - pin 1 (Motor Drive +), J3 – pin 6 (Motor Drive -)

**signal type:** TTL output

**notes:**

**explanation:** These module outputs provide the PWM drive signal for a DC servo motor. The PWM frequency is 31.25 KHz. The resolution of the PWM is a full eight bits, resulting in .0390625 volts per DAC unit. Rotational direction is determined by connecting the Motor Drive signals (Motor - and Motor +) to the appropriate terminals on the DC servo motor.

#### 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 oF command can be used to disable this signal.

#### Amplifier Enable Output

**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 Enable signal will be deactivated. This signal can also be deactivated by the Motor oF 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 oF 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:** The encoder power jumper JP3 sets the 'mid point' for the differential receiver

**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. By setting jumpers JP1 and JP2 appropriately, the plus signal inputs can be configured for single ended inputs.

#### **Auxiliary Encoder Inputs** (Phase A, 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 can be used for an auxiliary encoder.

#### **Encoder Power Output**

**connection point:** J3 pin 17

**signal type:** +5 VDC PC power supply output or +12 VDC PC power supply output

**notes:** The encoder power jumper JP3 selects +5VDC or +12VDC

**explanation:** This module pin provides a convenient supply voltage connection for the encoders. The jumper JP3 located on the module can be used to connect either the +5 or +12 volt supply to the Encoder Power pin. The setting of this jumper also selects the threshold voltage for the module's single ended phase and index encoder inputs. When JP1 is set for +5 volts, the threshold will be 2.5 volts, for +12 volts, the threshold will be +6 volts. The threshold voltage determines at what voltage the input changes between on and off.

**SUPPLY CONNECTIONS** (+5, +12, -12, GROUND) - These module pins provide access to the DCX supply voltages.

#### **Joystick Input (A/D Channel, 8 bit)**

**connection point:** J4 pin 1

**signal type:** 0.0 to +5V analog input

**notes:** used for joystick control

**explanation:** This input is used to implement manual jogging of the axis. See the description of **Jogging** in the **Motion Control** chapter.

#### **A/D +5 Volt Reference Output**

**connection point:** J4 pin 3

**signal type:** precision +5V reference voltage

**notes:** used for joystick control

**explanation:** This input is used to implement manual jogging of the axis. See the description of **Jogging** in the **Motion Control** chapter.

**Analog Ground**

**connection point:** J4 pin 4

**signal type:** analog ground for A/D conversion

**notes:** used for joystick control

**explanation:** This input is used to implement manual jogging of the axis. See the description of **Jogging** in the **Motion Control** chapter.

## DCX-MC210 Module connectors

### J3 connector pin-out (Motor command, encoders, and axis I/O)

Pin #	Description
1	PWM Motor Drive + (output, 500ma max.)
2	Encoder Power (+5VDC or +12VDC, see jumper JP3)
3	Primary Encoder Phase B+ (input)*
4	Primary Encoder Phase A+ (input)*
5	Ground
6	PWM Motor Drive - (output, 500ma max.)
7	Ext. Motor Power + (optional) ****
8	Primary Encoder Index + (input, active high)/ *** Ext. Motor Power - (optional) ****
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 Enable (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	Primary Encoder Phase A+ (input)*
17	Encoder Power (+5VDC or +12VDC, see jumper JP3)
18	Auxiliary Encoder Index - (input, active low)
19	Primary Encoder Phase A- (input)
20	Primary Encoder Phase B- (input)
21	Auxiliary Encoder Phase A
22	Auxiliary Encoder Phase B
23	Primary Encoder Phase B+ (input)*
24	Auxiliary Encoder Index+ (input, active high)
25	Primary 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.

\*\*\* Selected by JP5

\*\*\*\* For use when +12VDC supplied to DCX motherboard is not used for motor supply

Mating Connector: 26-pin dual-row IDC female, Circuit Assembly P/N CA-26IDS2-F-SPT or equivalent

### J4 connector pin-out (A/D channels for jogging)

Pin #	Description
1	Analog Input #1
2	Analog Input #2
3	+5 volt reference output
4	Analog ground



## DCX-MC210 Module Configuration Jumpers - configuration in **bold type** denotes default factory shipping configuration

### JP1 – Encoder type (single ended or differential)

<b>Pins</b>	<b>Description</b>
1 to 2 to 3	Single ended encoder, A, B, Z (three pin jumper provided)
open	<b>Differential encoder, A+, A-, B+, B-</b>

### JP2 – Encoder Index Active Level Select)

<b>Pins</b>	<b>Description</b>
1 to 2	Single ended Index, Z+ (Active high)
<b>2 to 3</b>	<b>Single ended Index, Z- (active low)</b>
open	Differential Index, Z+ and Z-

### JP3 – Encoder Power Select (+5VDC or +12 VDC)

<b>Pins</b>	<b>Description</b>
<b>1 to 2</b>	<b>+5 VDC encoder supply on J3 pin 17</b>
2 to 3	+12 VDC encoder supply on J3 pin 17

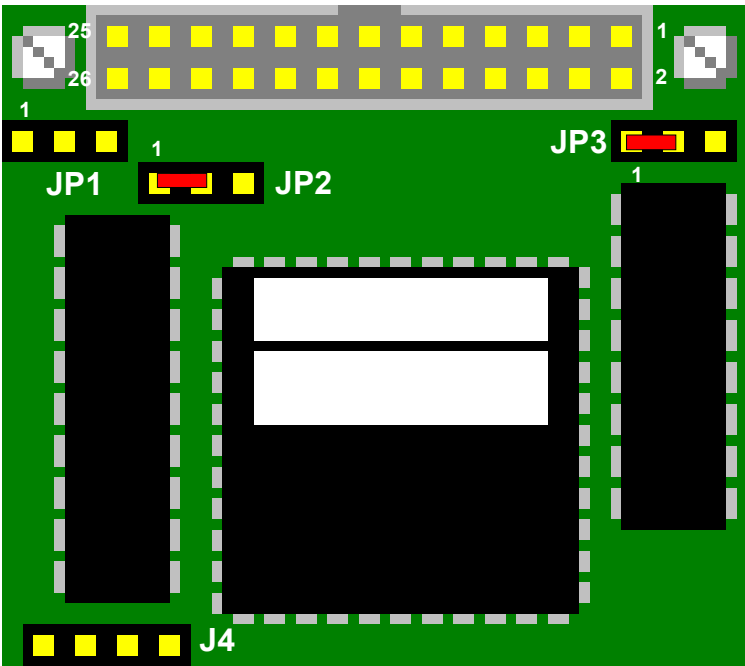
### JP4 – Motor Supply + (+12 VDC PC Power supply or external + supply)

<b>Pins</b>	<b>Description</b>
<b>1 to 2</b>	<b>+5 VDC encoder supply on J3 pin 17</b>
2 to 3 cut trace JP4 1 to 2 (bottom side)	Use external supply voltage (connected to J3 pin 7)

### JP5 – Motor Supply - (PC Ground or external supply -)

<b>Pins</b>	<b>Description</b>
<b>1 to 2</b>	<b>PC ground</b>
2 to 3 cut traces JP5 1 to 2 JP5 3 to 4 (bottom side)	Use external supply voltage – (connected to J3 pin 8)

DCX-MC210 Module Layout



## DCX-MC260 Stepper Motor Control Module

### SIGNAL DESCRIPTIONS:

#### Pulse and Direction Outputs

**connection point:** J3 - pin 3 (Direction/CW), J3 – pin 4 (Pulse/CCW)

**signal type:** open collector driver

**notes:** external pull-up required

**explanation:** In the control of a stepper motor, the two primary control signals are Pulse and Direction (or CW Pulse and CCW Pulse). These signals are connected to the external stepper motor driver that supplies current to the motor windings. In order for the stepper module to move the motor one step, a pulse is generated on one of these signals.

The motor driver should advance the motor by one increment for each pulse. The motor may advance a full step, a half step, or a micro step. This is determined by the mode of the stepper motor driver. The Pulse signal is normally high, and goes low at the beginning of a step. It stays low for one half the step period (ie. the time before the next pulse), and then goes back high. When it is time for the next step, the signal will go low again. The Direction signal selects which direction the motor will move. When this signal is high, the stepper controller will decrement the current position for every step taken, and when the signal is low, it will increment the current position for every step taken. Both of these signals have high current open collector drivers on the module and are suitable for direct connection to optically isolated inputs commonly found on stepper motor drivers. Because of the characteristics of open collector drivers, no voltages will be present on these output signals unless pull-up resistors are connected to them.

The Output Mode command can be used to change the operation of these signals to CW and CCW Pulse. In this mode, pulses will be generated on the CW Pulse output when the current position is increasing, and on the CCW Pulse output when the current position is decreasing.

#### Stopped (motion complete) Output

**connection point:** J3 - pin 7

**signal type:** TTL output

**notes:**

**explanation:** The Stopped signal is a status output from the stepper module, indicating when the motor is stepping. At the beginning of a motion, the Stopped signal is brought high. It will continue to stay high for the duration of that motion. At the end of the motion, the Stopped signal is brought low again. The high to low transition on Stopped indicates when the motion is over, and the low level indicates that the motor is no longer moving.

The Stopped signal may be used as a motion complete indication, it may also control the power selection for the stepper motor power driver, switching automatically between a high current while stepping, and low current while stopped. This will reduce power dissipation of the motor when it isn't moving.

#### Limit Positive and Limit Negative Inputs

**connection point:** J3 - pin 8 (Limit Positive), J3 - pin 9 (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 oF commands respectively. See the description of **Motion Limits** in the **Motion Control** chapter.

#### Home Input

**connection point:** J3 - pin 13

**signal type:** TTL input

**notes:** 4.7K pull up resistor is connected to the +5V logic supply

**explanation:** This input is used to set the stepper motors zero position. It is typically connected to a switch that is activated at a fixed position in the motor's range of motion.

#### Full/Half Step Output

**connection point:** J3 - pin 14

**signal type:** TTL output

**notes:** 2ma sink/source

**explanation:** If the motor driver that this module is controlling has a digital input to select between full and half step modes, this module output is used to select between the modes. This output will be set low by the Step Full (SF) command or high by the Step Half (SH) commands.

#### Full/Half Current Output

**connection point:** J3 - pin 15

**signal type:** TTL output

**notes:** 2ma sink/source

**explanation:** If the motor driver that the module is controlling has a digital current control signal input, this module output can be used to control the motor drivers output current. This output will be set low by the Full Current (FC) command or high by the Half Current (HC) commands. Normally a stepper motor driver will be set for full current while it is moving and half current while stationary, to reduce motor heating.

#### Motor On Output

**connection point:** J3 - pin 16

**signal type:** TTL output

**notes:** 2ma sink/source

**explanation:** This module output will go low when the Motor oN (MN) command is issued. ***It will go high when the Motor oF (MF) command is issued, the controller is reset, or a limit switch input is activated.*** This signal can be connected to the enable signal input of the stepper driver so that it can be disabled by issuing commands to the DCX.

#### Null Input

**connection point:** J3 - pin 17

**signal type:** TTL input

**notes:** 4.7K pull up resistor is connected to the +5V logic supply

**explanation:** In order to switch from micro stepping to full stepping without the motor shifting position, the motor should be micro stepped to the "Null" Position. This is the position where the output of the amplifier will not change if it is switched between full and micro stepping. If the stepper amplifier provides an output signal that indicates when the motor is at a null position, the DCX can monitor this signal on the Null Position input of the module.

**Auxiliary Encoder Inputs** (Phase A & A+, Phase B & B+, Index-)**connection point:** see pin-out table**signal type:** TTL or Differential driver output (-7V to +7V)**notes:****explanation:** - These input signals can be used for an auxiliary encoder.**Encoder Coarse Home Input****connection point:** J3 - pin 23**signal type:** TTL input**notes:** 4.7K pull up resistor is connected to the +5V logic supply**explanation:** This input is used to 'home' the auxiliary encoder by qualifying the index mark. It is typically connected to a switch that is activated at a fixed position in the motors motion range. See the description of **Homing an Axis** in the **Motion Control** chapter.**SUPPLY CONNECTIONS** (+5, +12, -12, GROUND) - These module pins provide access to the DCX supply voltages.**Joystick Input (A/D Channel, 8 bit)****connection point:** J4 pin 1**signal type:** 0.0 to +5V analog input**notes:** used for joystick control**explanation:** This input is used to implement manual jogging of the axis. This signal is also available on connector J3 pin 10. See the description of **Jogging** in the **Motion Control** chapter.**A/D +5 Volt Reference Output****connection point:** J4 pin 3**signal type:** precision +5V reference voltage**notes:** used for joystick control**explanation:** This input is used to implement manual jogging of the axis. See the description of **Jogging** in the **Motion Control** chapter.**Analog Ground****connection point:** J4 pin 4**signal type:** analog ground for A/D conversion**notes:** used for joystick control**explanation:** This input is used to implement manual jogging of the axis. See the description of **Jogging** in the **Motion Control** chapter.

## DCX-MC260 Module connectors

### J3 connector pin-out (Motor command, encoders, and axis I/O)

Pin #	Description
1	Ground
2	+5 VDC
3	Direction or CW Pulse (output, active low, open collector driver, 100 ma max.)*
4	Pulse or CCW Pulse (output, active low, open collector driver, 100 ma max.)*
5	Reserved
6	Reserved
7	Stopped (output, high while moving, TTL level)**
8	Limit Positive (input, active low, with 4.7K ohm pull-up to +5V)
9	Limit Negative (input, active low, with 4.7K ohm pull-up to +5V)
10	Jog Input (connected to J4 pin 1)
11	Reserved
12	Reserved
13	Home (input, active low, with 4.7K ohm pull-up to +5V)
14	Full/Half Step (output, low when full stepping, TTL level)**
15	Full/Half Current (output, low when moving, TTL level)**
16	Motor On (output, low when motor on, TTL level)**
17	Null Position (input, active low, with 4.7K ohm pull-up to +5V)
18	Auxiliary Encoder Phase A+ (input)
19	Auxiliary Encoder Phase A- (input)
20	Auxiliary Encoder Phase B+ (input)
21	Auxiliary Encoder Phase B- (input)
22	Auxiliary Encoder Index- (input, active low)
23	Auxiliary Encoder Coarse Home (input, active low, with 4.7K ohm pull-up to +5V)
24	+12 VDC
25	-12 VDC
26	Ground

\* These signals default to DIRECTION and PULSE, use Output Mode command to change to CW and CCW PULSE.

\*\* These signals are not suitable for directly driving optically isolated inputs.

Mating Connector: 26-pin dual-row IDC female, Circuit Assembly P/N CA-26IDS2-F-SPT or equivalent

### J4 connector pin-out (A/D channels for jogging)

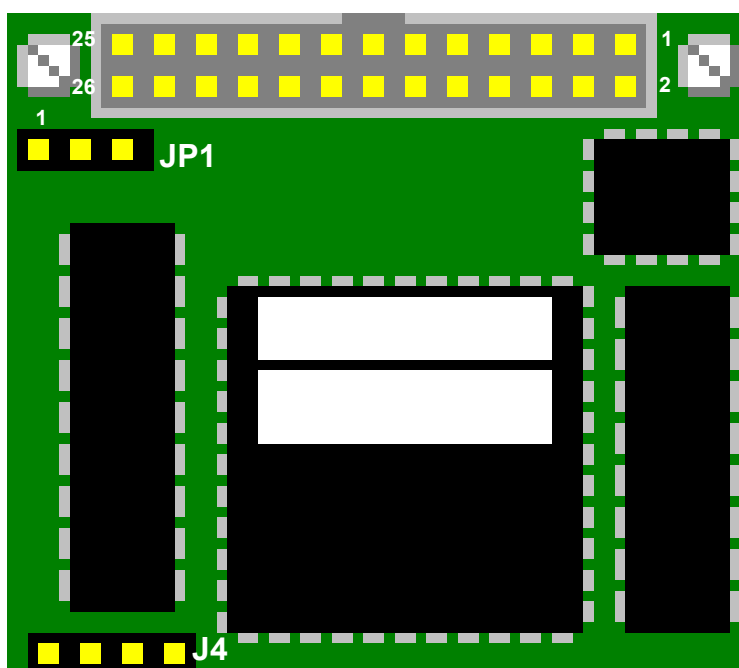
Pin #	Description
1	Analog Input #1
2	Analog Input #2
3	+5 volt reference output
4	Analog ground

## DCX-MC260 Module Configuration Jumpers - configuration in **bold type** denotes default factory shipping configuration

### JP1 – Encoder type (single ended or differential)

<i><b>Pins</b></i>	<i><b>Description</b></i>
1 to 2 to 3	Single ended encoder, A, B, Z (three pin jumper provided)
<b>open</b>	<b>Differential encoder, A+, A-, B+, B-</b>

### DCX-MC260 Module Layout



## DCX-MC400 Digital I/O Module

### DCX-MC400 Electrical Specifications

<b>Parameter</b>	<b>Min.</b>	<b>Max</b>	<b>Unit</b>
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

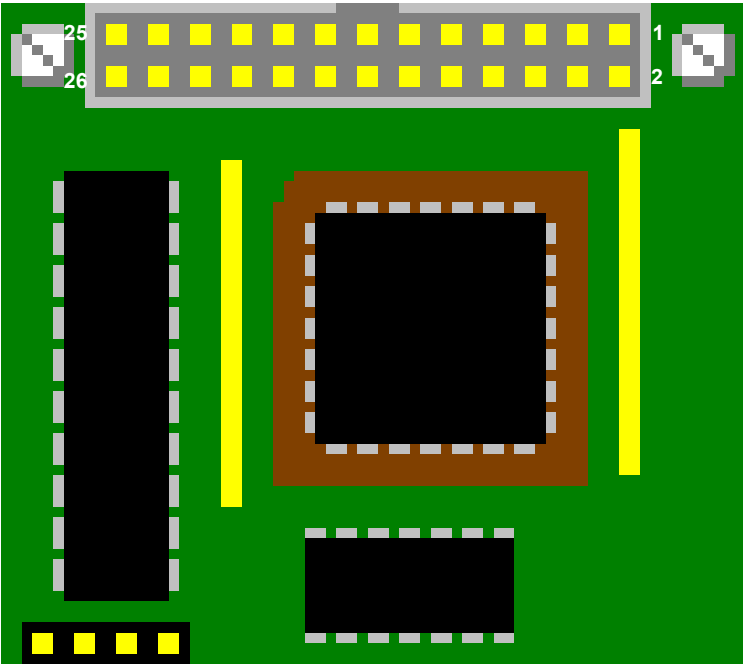
### J3 connector pin-out

<b>Pin #</b>	<b>Description</b>
1	Digital I/O channel #1
2	Digital I/O channel #1
3	Digital I/O channel #1
4	Digital I/O channel #1
5	Digital I/O channel #1
6	Digital I/O channel #1
7	Digital I/O channel #1
8	Digital I/O channel #1
9	Digital I/O channel #1
10	Digital I/O channel #1
11	Digital I/O channel #1
12	Digital I/O channel #1
13	Digital I/O channel #1
14	Digital I/O channel #1
15	Digital I/O channel #1
16	Digital I/O channel #1
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 CA-26IDS2-F-SPT or equivalent



**DCX-MC400 Module layout**



## DCX-MC500/510/520 Analog I/O Module

### J3 connector pin-out

<b>Pin #</b>	<b>Description</b>
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

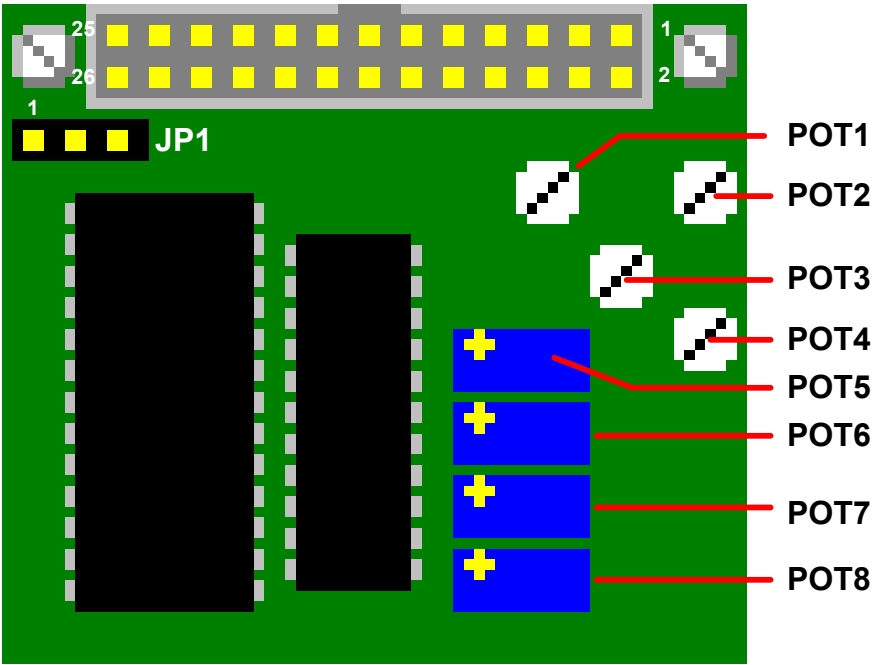
Mating Connector: 26-pin dual-row IDC female, Circuit Assembly P/N CA-26IDS2-F-SPT or equivalent

### 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)

<b>Pins</b>	<b>Description</b>
1 to 2	Use external reference (supplied by user on J3 pin 18)
<b>2 to 3</b>	<b>Use the on board +5 VDC reference</b>

**DCX-MC500 Module layout**



## DCX-MF300 – RS-232 Interface Module

### J3 connector pin-out

Pin #	Description
1	*
2	*
3	Receive (Maps to DB25 pin 2)
4	*
5	Transmit (Maps to DB25 pin 3)
6	*
7	Clear to Send (Maps to DB25 pin 4)
8	*
9	Request to Send (Maps to DB25 pin 5)
10	*
11	Data Set Ready (Maps to DB25 pin 6)
12	*
13	Ground (Maps to DB25 pin 7)
14	Data Terminal Ready (Maps to DB25 pin 20)
15	Data Carrier Detect (Maps to DB25 pin 8)
16	*
17	*
18	*
19	*
20	*
21	*
22	*
23	*
24	*
25	*
26	*

\* No connect

Mating Connector: 26-pin dual-row IDC female, Circuit Assembly P/N CA-26IDS2-F-SPT or equivalent

### DCE/DTE (jumpers JP3 – JP10) configuration

#### DCE (factory default)

**JP3 - 1 to 2**

**JP4 - 1 to 2**

**JP6 - 1 to 2**

**JP7 - 2 to 3**

**JP9 - 1 to 2**

**JP10 - 1 to 2**

JP5 - 1 to 2 or

**JP8 - 1 to 2**

#### DTE

JP3 - 2 to 3

JP4 - 2 to 3

JP5 - 2 to 3

JP8 - 2 to 3

JP10 - 1 to 2

JP6 - 2 to 3 or

JP7 - 2 to 3 or

JP9 - 2 to 3

(See the layout diagram at end of this appendix)

**DCX-MF300 Module Configuration Jumpers** - configuration in **bold type** denotes default factory shipping configuration

**JP1 – Baud Rate select**

<b>Pins</b>	<b>Description</b>
1 to 2	300 baud
3 to 4	1200 baud
5 to 6	2400 baud
7 to 8	4800 baud
<b>9 to 10</b>	<b>9600 baud</b>
11 to 12	19200 baud
13 to 14	Enable network

**JP2 – Handshake/Network Select**

<b>Pins</b>	<b>Description</b>
<b>1 to 3, 2 to 4</b>	<b>Hardware handshaking</b>
1 to 2, 3 to 4	Networking (multiple devices on a serial line)

**JP3 – Receive pin select**

<b>Pins</b>	<b>Description</b>
<b>1 to 2</b>	<b>Data input on J3 pin 3</b>
2 to 3	Data input on J3 pin 3

**JP4 – Transmit pin select**

<b>Pins</b>	<b>Description</b>
<b>1 to 2</b>	<b>Data output on J3 pin 5</b>
2 to 3	Data input on J3 pin 3

**JP5 – Pin 7 select**

<b>Pins</b>	<b>Description</b>
<b>1 to 2</b>	<b>DCE</b>
2 to 3	DTE

**JP6 – Pin 9 select**

<b>Pins</b>	<b>Description</b>
<b>1 to 2</b>	<b>DCE</b>
2 to 3	DTE

**JP7 – Pin 11 select**

<b>Pins</b>	<b>Description</b>
1 to 2	DTE
2 to 3	DCE

**DCX-MF300 Module Configuration Jumpers (cont.)** - configuration in **bold type** denotes default factory shipping configuration

**JP8 – Pin 14 select**

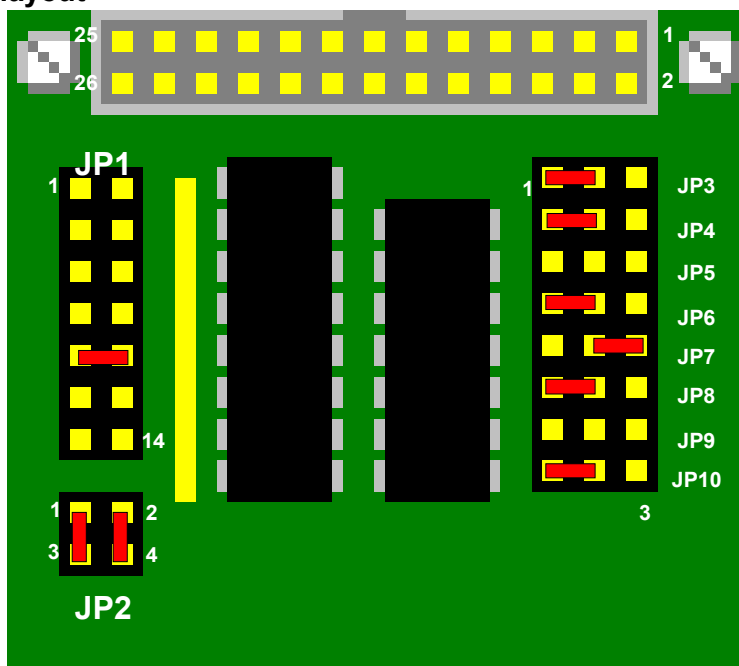
<b>Pins</b>	<b>Description</b>
1 to 2	DCE
2 to 3	DTE

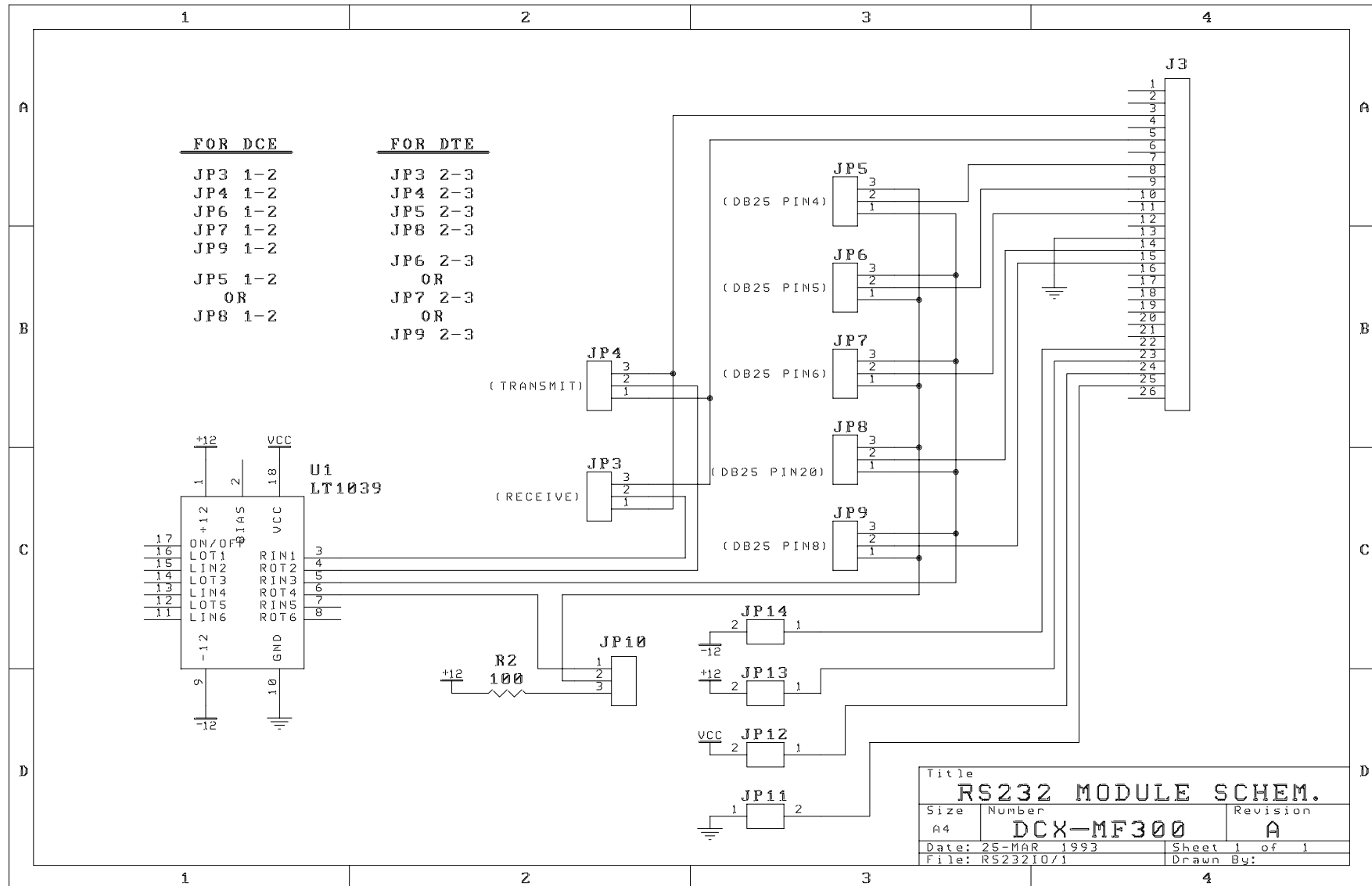
**JP9 – Pin 15 select**

<b>Pins</b>	<b>Description</b>
1 to 2	DCE
2 to 3	DTE

**JP10 – Ready select**

<b>Pins</b>	<b>Description</b>
1 to 2	<b>DCX Ready signal</b>
2 to 3	100 ohm pull-up to +12V (This signal goes to JP5-3, JP6-1, JP7-1, JP8-3, JP9-1)

**DCX-MF300 Module layout**



## DCX-MF310 IEEE-488 Interface Module

**J3 connector pin-out** - The signals are arranged so that the connection to a standard IEEE-488 connector will be straight through a ribbon cable.

<b>Pin #</b>	<b>Description</b>
1	IEEE-488: D101
2	IEEE-488: D105
3	IEEE-488: D102
4	IEEE-488: D106
5	IEEE-488: D103
6	IEEE-488: D107
7	IEEE-488: D104
8	IEEE-488: D108
9	IEEE-488: E01
10	IEEE-488: REN
11	IEEE-488: DAV
12	IEEE-488: Ground
13	IEEE-488: NRFD
14	IEEE-488: Ground
15	IEEE-488: NDAC
16	IEEE-488: Ground
17	IEEE-488: IFC
18	IEEE-488: Ground
19	IEEE-488: SRQ
20	IEEE-488: Ground
21	IEEE-488: ATN
22	IEEE-488: Ground
23	IEEE-488: Shield
24	IEEE-488: GND
25	IEEE-488: Not used
26	IEEE-488: Not used

Mating Connector: 26-pin dual-row IDC female, Circuit Assembly P/N CA-26IDS2-F-SPT or equivalent



---

**DCX-MF310 Module Configuration Jumpers** - configuration in **bold type** denotes default factory shipping configuration**JP1 – Baud Rate select**

<b><i>Pins</i></b>	<b><i>Description</i></b>
1 to 2	Open collector drivers
<b>open</b>	<b>Push –Pull drivers</b>

Note: Push-pull used for high speed bus operation

**JP2 – Select cable shield to ground**

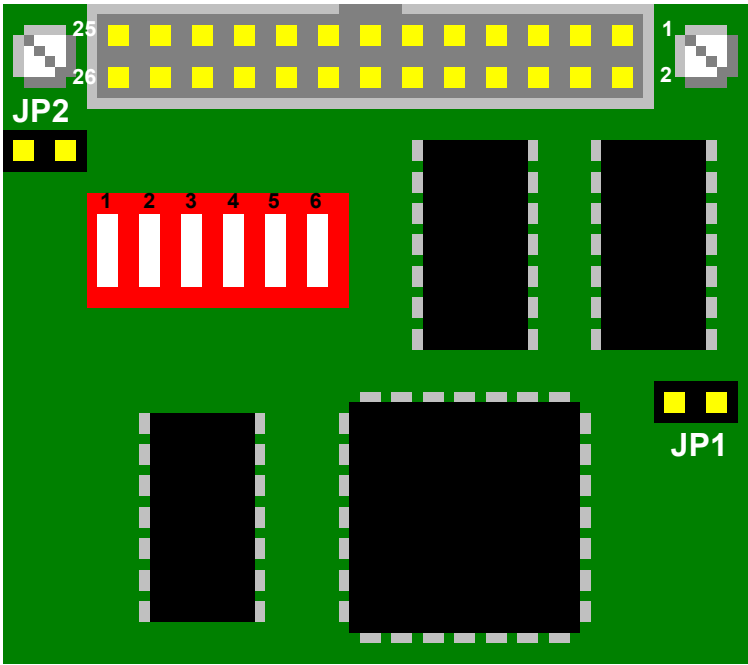
<b><i>Pins</i></b>	<b><i>Description</i></b>
1 to 2	Tie cable shield to DCX ground
<b>open</b>	

Note: Cable shield should only be grounded at one point

**IEEE-488 Bus Address Selection**

<b>SW6</b>	<b>SW5</b>	<b>SW4</b>	<b>SW3</b>	<b>SW2</b>	<b>SW1</b>		<b>Listen</b>	<b>Talk</b>
Off	Off	Off	Off	Off	Off		" " (20h)	" " (40h)
Off	Off	Off	Off	Off	On		"!" (21h)	"A" (41h)
Off	Off	Off	Off	On	Off		"'" (22h)	"B" (42h)
Off	Off	Off	Off	On	On		"#" (23h)	"C" (43h)
Off	Off	Off	On	Off	Off		"\$" (24h)	"D" (44h)
Off	Off	Off	On	Off	On		"%" (25h)	"E" (45h)
Off	Off	Off	On	On	Off		"&" (26h)	"F" (46h)
Off	Off	Off	On	On	On		"'" (27h)	"G" (47h)
Off	Off	On	Off	Off	Off		"(" (28h)	"H" (48h)
Off	Off	On	Off	Off	On		")" (29h)	"I" (49h)
Off	Off	On	Off	On	Off		"*" (2Ah)	"J" (4Ah)
Off	Off	On	Off	On	On		"+" (2Bh)	"K" (4Bh)
Off	Off	On	On	Off	Off		"," (2Ch)	"L" (4Ch)
Off	Off	On	On	Off	On		"-" (2Dh)	"M" (4Dh)
Off	Off	On	On	On	Off		"." (2Eh)	"N" (4Eh)
Off	Off	On	On	On	On		"/" (2Fh)	"O" (4Fh)
Off	On	Off	Off	Off	Off		"0" (30h)	"P" (50h)
Off	On	Off	Off	Off	On		"1" (31h)	"Q" (51h)
Off	On	Off	Off	On	Off		"2" (32h)	"R" (52h)
Off	On	Off	Off	On	On		"3" (33h)	"S" (53h)
Off	On	Off	On	Off	Off		"4" (34h)	"T" (54h)
Off	On	Off	On	Off	On		"5" (35h)	"U" (55h)
Off	On	Off	On	On	Off		"6" (36h)	"V" (56h)
Off	On	Off	On	On	On		"7" (37h)	"W" (57h)
Off	On	On	Off	Off	Off		"8" (38h)	"X" (58h)
Off	On	On	Off	Off	On		"9" (39h)	"Y" (59h)
Off	On	On	Off	On	Off		": (3Ah)	"Z" (5Ah)
Off	On	On	Off	On	On		"[" (3Bh)	"[" (5Bh)
Off	On	On	On	Off	Off		"<" (3Ch)	"\" (5Ch)
Off	On	On	On	Off	On		"=" (3Dh)	"]" (5Dh)
Off	On	On	On	On	Off		">" (3Eh)	" " (5Eh)
Off	On	On	On	On	On		"?" (3Fh)	" " (5Fh)

**DCX-MF310 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.

<b>Pin #</b>	<b>Description</b>
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 CA-26IDS2-F-SPT 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.

<b>Pin #</b>	<b>Description</b>
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 CA-26IDS2-F-SPT or equivalent

## DCX-BF022 Configuration Jumpers - configuration in **bold type** denotes default factory shipping configuration

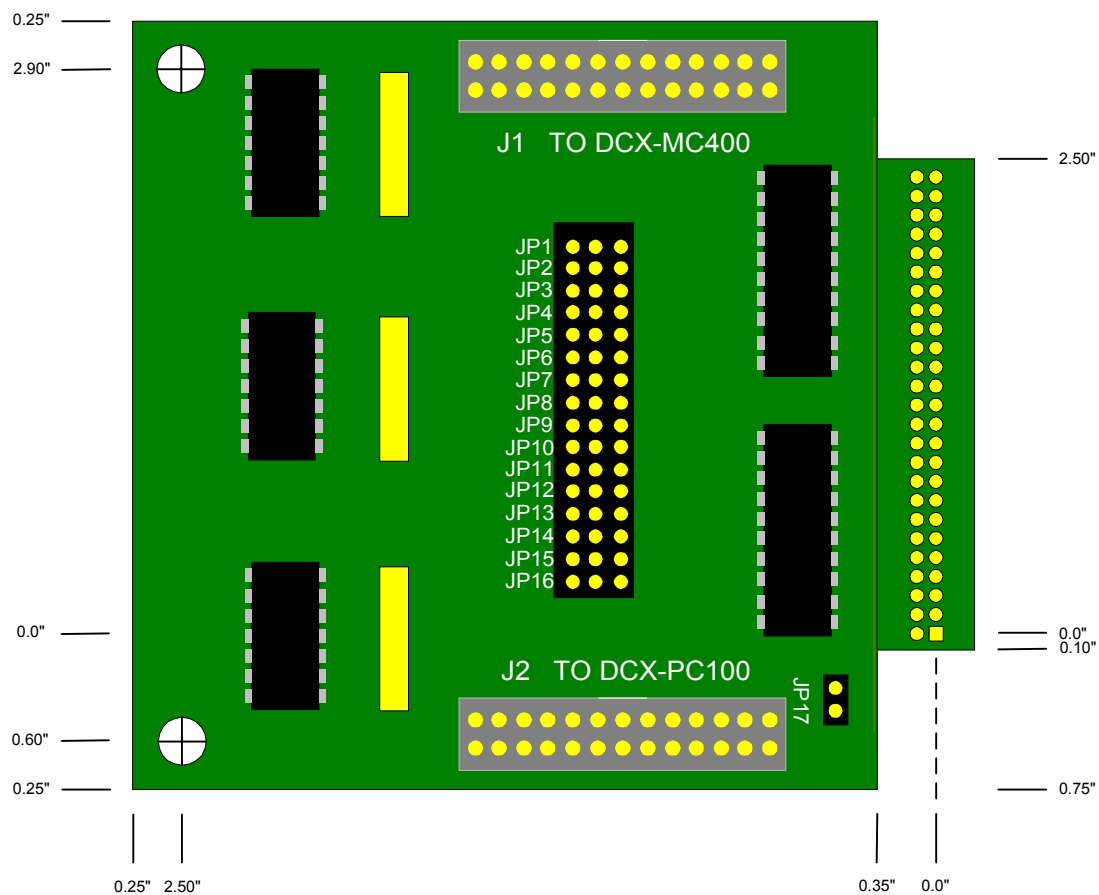
### JP1 – JP16 Configure Digital channel as Input or Output

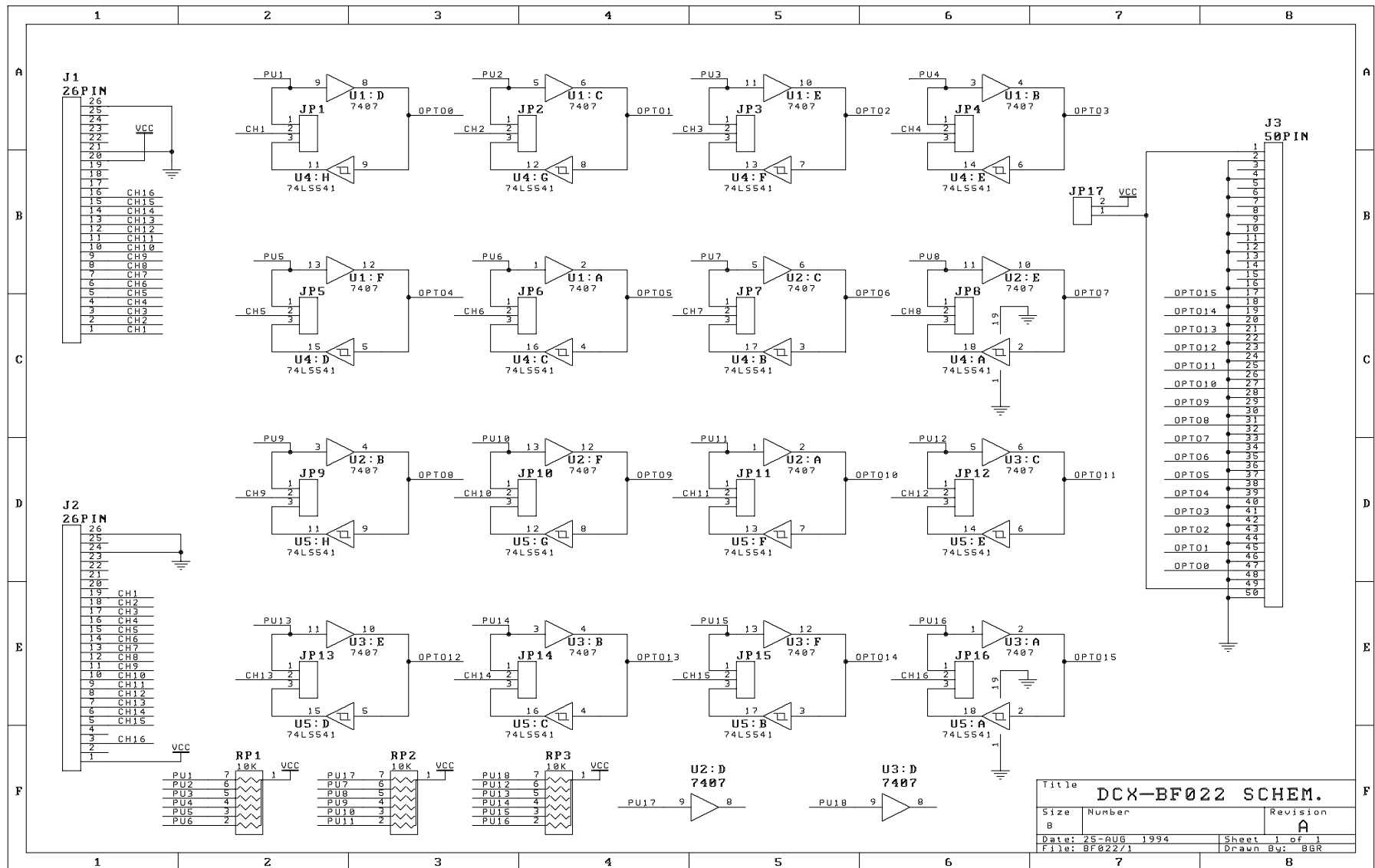
<b>Pins</b>	<b>Description</b>
<b>1 to 2</b>	<b>Configure channel as Output</b>
2 to 3	Configure channel as an Input

### JP17 – Select Relay Rack supply source

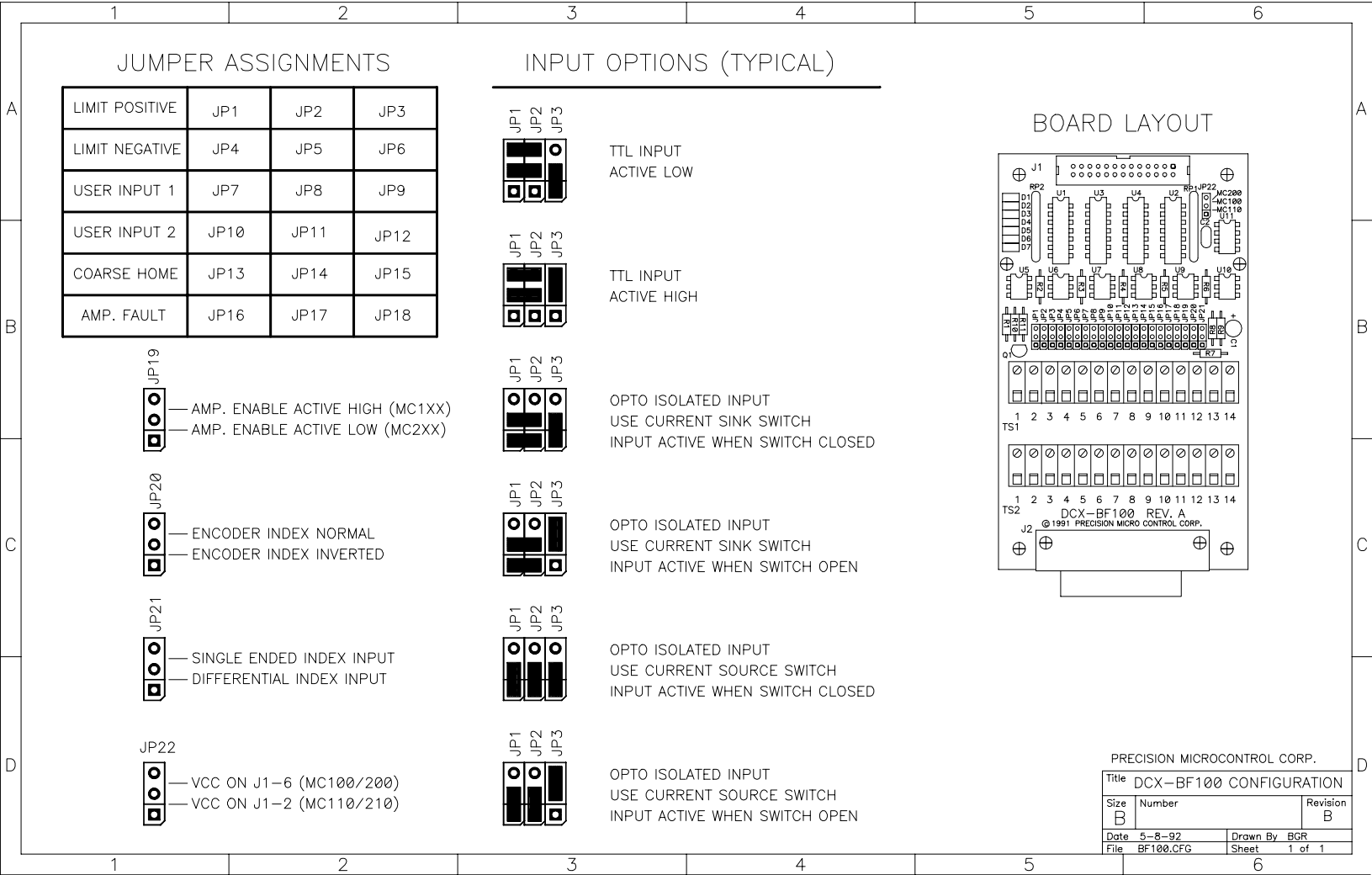
<b>Pins</b>	<b>Description</b>
<b>1 to 2</b>	<b>DCX provides +5 VDC Relay Rack supply</b>
2 to 3	Relay Rack has separate +5 VDC supply

### DCX-BF022 Interface layout





DCX-BF100 Servo Module Interconnect Board





**DCX-BF100 to DCX-MC200 Connections:****Connector J1**

<b>Pin</b>	<b>Description</b>
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 Enable
12	User Input #1
13	User Input #2
14	Limit +
15	Limit -
16	Prim. Encoder Phase A+
17	Encoder Power
18	Aux. Encoder Index-
19	Prim. Encoder Phase A-
20	Prim. Encoder Phase B-
21	Aux. Encoder Phase A
22	Aux. Encoder Phase B
23	Prim. Encoder Phase B+
24	Aux. Encoder Index+
25	Prim. Encoder Index-
26	Ground

**Connector J2**

<b>Pin</b>	<b>Description</b>
1	Analog Ground
2	+12 VDC
3	Ground
4	Opto Supply
5	Coarse Home
6	Amplifier Enable
7	User Input #2
8	Limit -
9	Encoder Power
10	Prim. Encoder Phase A-
11	Aux. Encoder Phase A
12	Prim. Encoder Phase B+
13	Prim. Encoder Index-
14	Analog Command output
15	-12 VDC
16	+5 VDC
17	Prim. Encoder Index+
18	Amplifier Fault
19	User Input #1
20	Limit +
21	Prim. Encoder Phase A+
22	Aux. Encoder Index-
23	Prim. Encoder Phase B-
24	Aux. Encoder Phase B
25	Aux. Encoder Index+

Note: Set MC200 module for single ended encoder index input (ie. connect jumper JP2 pins 2 and 3)

**Terminal strip TS1**

<b>Pin</b>	<b>Description</b>
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	User Input #1
10	User Input #2
11	Limit +
12	Limit -
13	Opto Supply
14	Ground

**Terminal strip TS2**

<b>Pin</b>	<b>Description</b>
1	Shield
2	Encoder Power
3	Prim. Encoder Phase A+
4	Prim. Encoder Phase A-
5	Prim. Encoder Phase B+
6	Prim. Encoder Phase B-
7	Prim. Encoder Index+
8	Prim. Encoder Index-
9	Aux. Encoder Phase A
10	Aux. Encoder Phase A
11	Aux. Encoder Index+
12	Aux. Encoder Index-
13	+5 VDC
14	Ground

**DCX-BF100 to DCX-MC210 Connections:****Connector J1**

<b>Pin</b>	<b>Description</b>
1	PWM Motor Drive +
2	Encoder Power
3	Prim. Encoder Phase B+
4	Prim. Encoder Phase A+
5	Ground
6	PWM Motor Drive -
7	Ext. Motor Power +
8	Prim. Enc. Index+/Ext -
9	Coarse Home
10	Amplifier Fault
11	Amplifier Enable
12	Reserved
13	Reserved
14	Limit +
15	Limit -
16	Prim. Encoder Phase A+
17	Encoder Power
18	Aux. Encoder Index-
19	Prim. Encoder Phase A-
20	Prim. Encoder Phase B-
21	Aux. Encoder Phase A
22	Aux. Encoder Phase B
23	Prim. Encoder Phase B+
24	Aux. Encoder Index+
25	Prim. Encoder Index-
26	Ground

**Connector J2**

<b>Pin</b>	<b>Description</b>
1	PWM Motor Drive +
2	Prim. Encoder Phase B+
3	Ground
4	Opto Supply
5	Coarse Home
6	Amplifier Enable
7	No connect
8	Limit -
9	Encoder Power
10	Prim. Encoder Phase A-
11	Aux. Encoder Phase A
12	Prim. Encoder Phase B+
13	Prim. Encoder Index-
14	Encoder Power
15	Prim. Encoder Phase A+
16	PWM Motor Drive +
17	Prim. Encoder Index+
18	Amplifier Fault
19	No connect
20	Limit +
21	Prim. Encoder Phase A+
22	Aux. Encoder Index-
23	Prim. Encoder Phase B-
24	Aux. Encoder Phase B
25	Aux. Encoder Index+

Note: Set MC210 module for single ended encoder index input (ie. connect jumper JP2 pins 2 and 3)

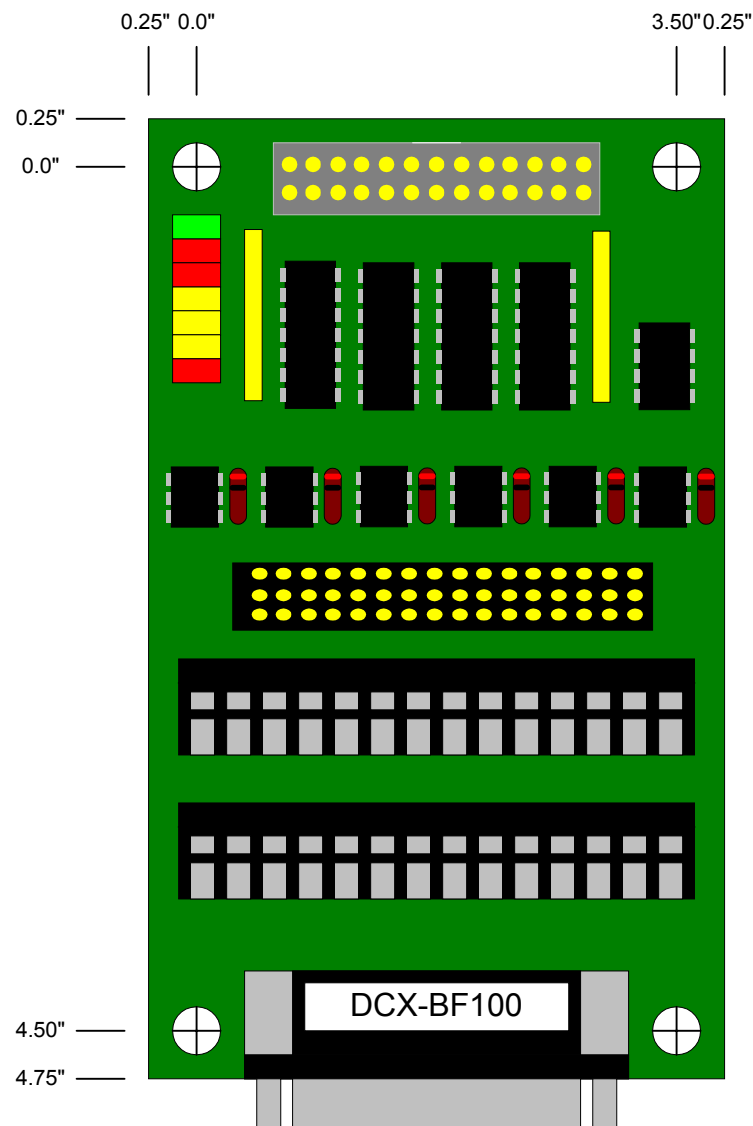
**Terminal strip TS1**

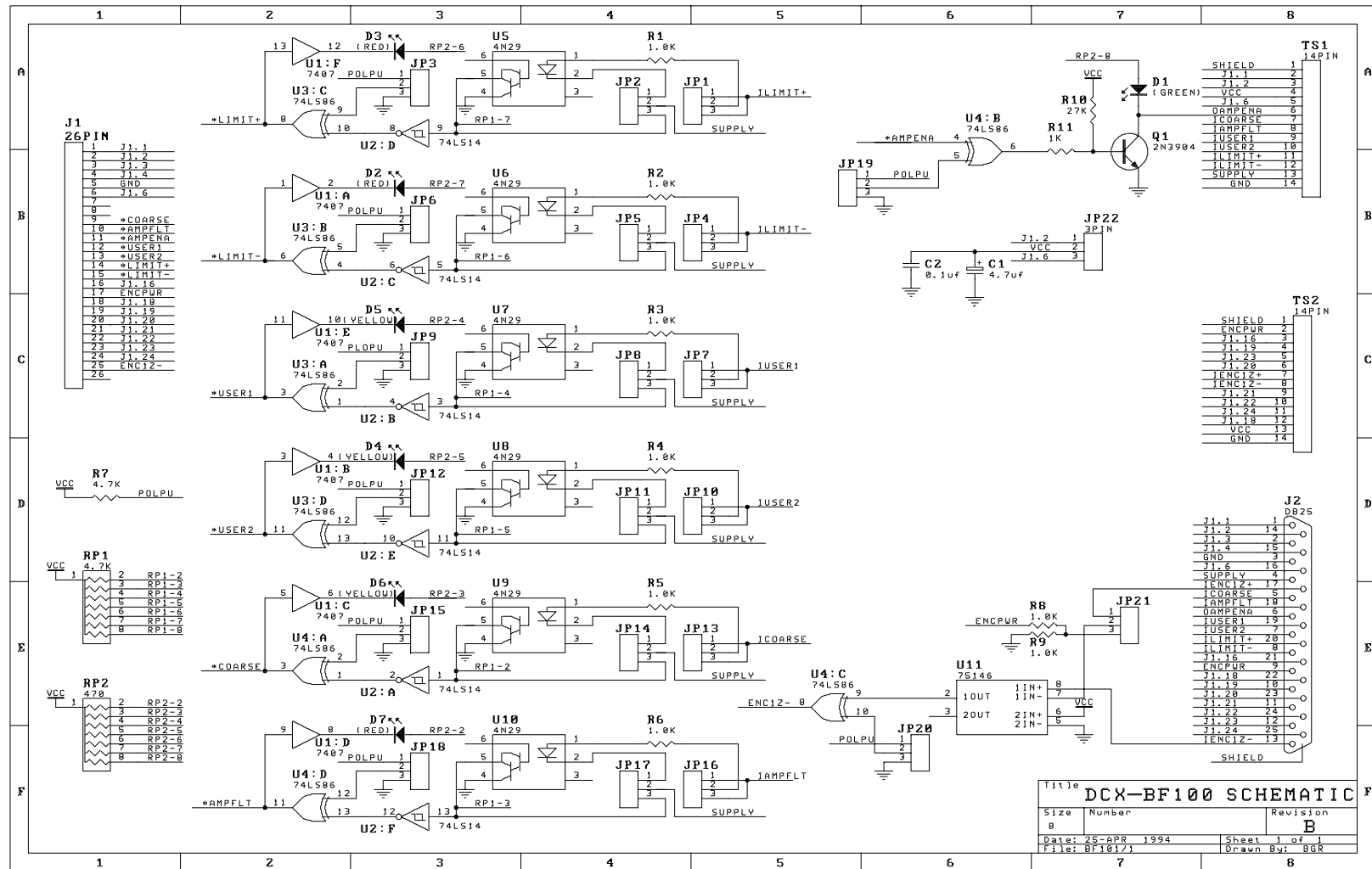
<b>Pin</b>	<b>Description</b>
1	Shield
2	PWM Motor Drive +
3	Encoder Power
4	+5 VDC
5	PWM Motor Drive -
6	Amplifier Enable
7	Coarse Home
8	Amplifier Fault
9	No connect
10	No connect
11	Limit +
12	Limit -
13	Opto Supply
14	Ground

**Terminal strip TS2**

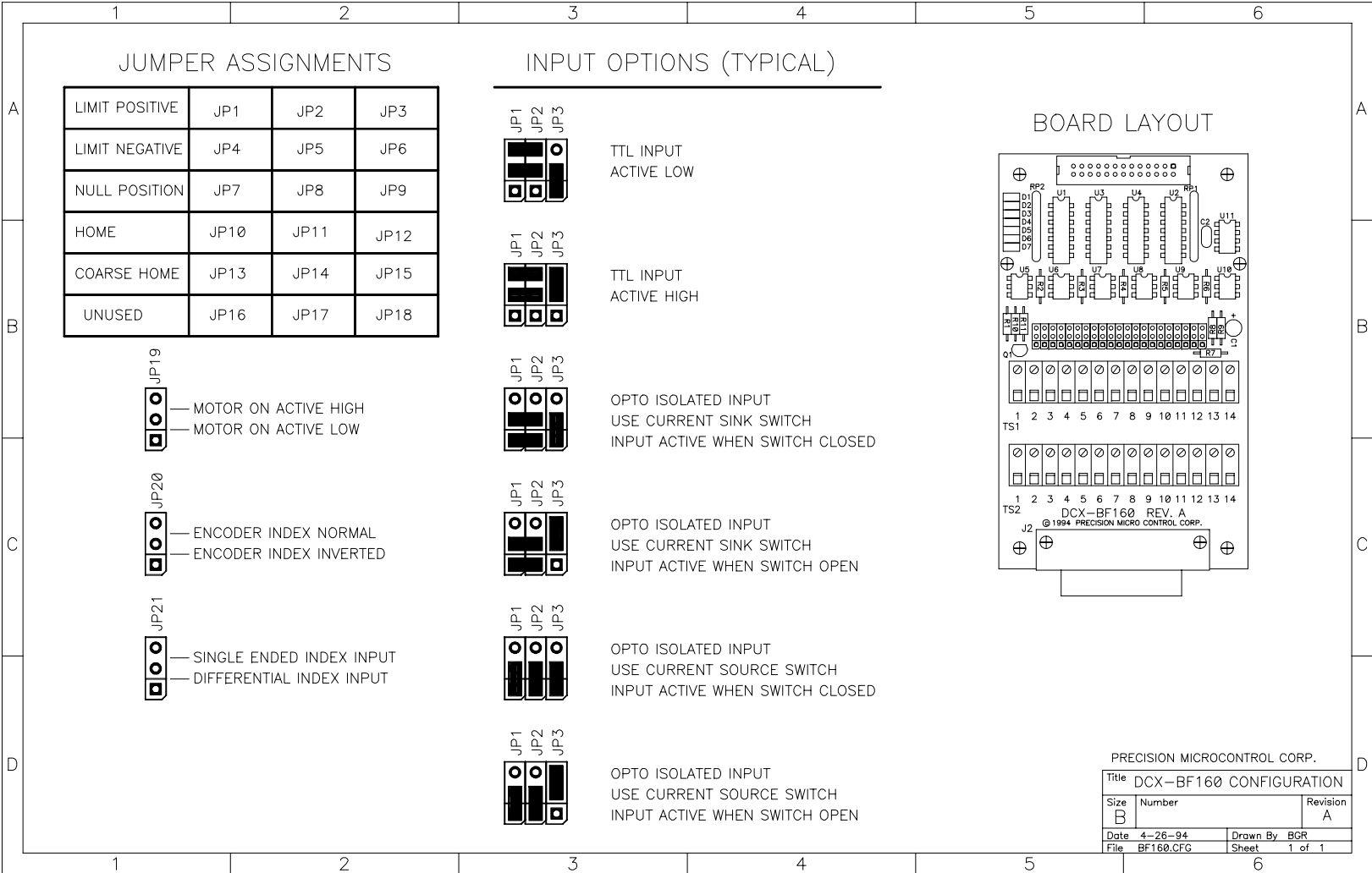
<b>Pin</b>	<b>Description</b>
1	Shield
2	Encoder Power
3	Prim. Encoder Phase A+
4	Prim. Encoder Phase A-
5	Prim. Encoder Phase B+
6	Prim. Encoder Phase B-
7	Prim. Encoder Index+
8	Prim. Encoder Index-
9	Aux. Encoder Phase A
10	Aux. Encoder Phase A
11	Aux. Encoder Index+
12	Aux. Encoder Index-
13	+5 VDC
14	Ground

**DCX-BF100 Interface layout**





DCX-BF160 Stepper Module Interconnect Board



**DCX-BF160 to DCX-MC260 Connections:****Connector J1**

<b>Pin</b>	<b>Description</b>
1	Ground
2	+5 VDC
3	Direction/CW Pulse
4	Pulse/CCW Pulse
5	Reserved
6	Reserved
7	Stopped
8	Limit +
9	Limit -
10	Jog
11	No connect
12	No connect
13	Home
14	Full/Half Step
15	Full/Half Current
16	Motor On
17	Null Position
18	Aux. Encoder Phase A+
19	Aux. Encoder Phase A-
20	Aux. Encoder Phase B+
21	Aux. Encoder Phase B-
22	Aux. Encoder Index
23	Aux. Enc. Coarse Home
24	+12 VDC
25	-12 VDC
26	Ground

**Connector J2**

<b>Pin</b>	<b>Description</b>
1	Ground
2	Direction/CW Pulse
3	Reserved
4	Stopped
5	Limit -
6	Opto Supply
7	Home
8	Full/Half Current
9	Null Position
10	Aux. Encoder Phase A-
11	Aux. Encoder Phase B-
12	Aux. Enc. Coarse Home
13	-12 VDC
14	+5 VDC
15	Pulse/CCW Pulse
16	Reserved
17	Limit +
18	Jog
19	Aux. Encoder Index+
20	Full/Half Step
21	Motor On
22	Aux. Encoder Phase A+
23	Aux. Encoder Phase B+
24	Aux. Encoder Index-
25	+12 VDC

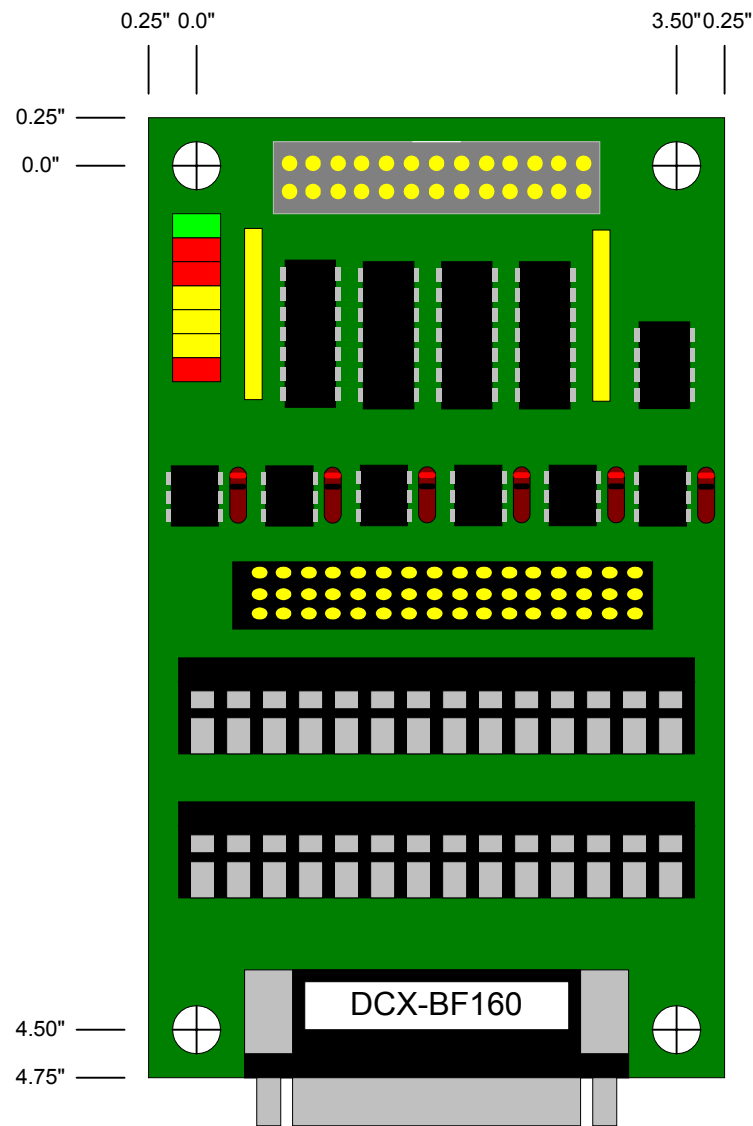
**Terminal strip TS1**

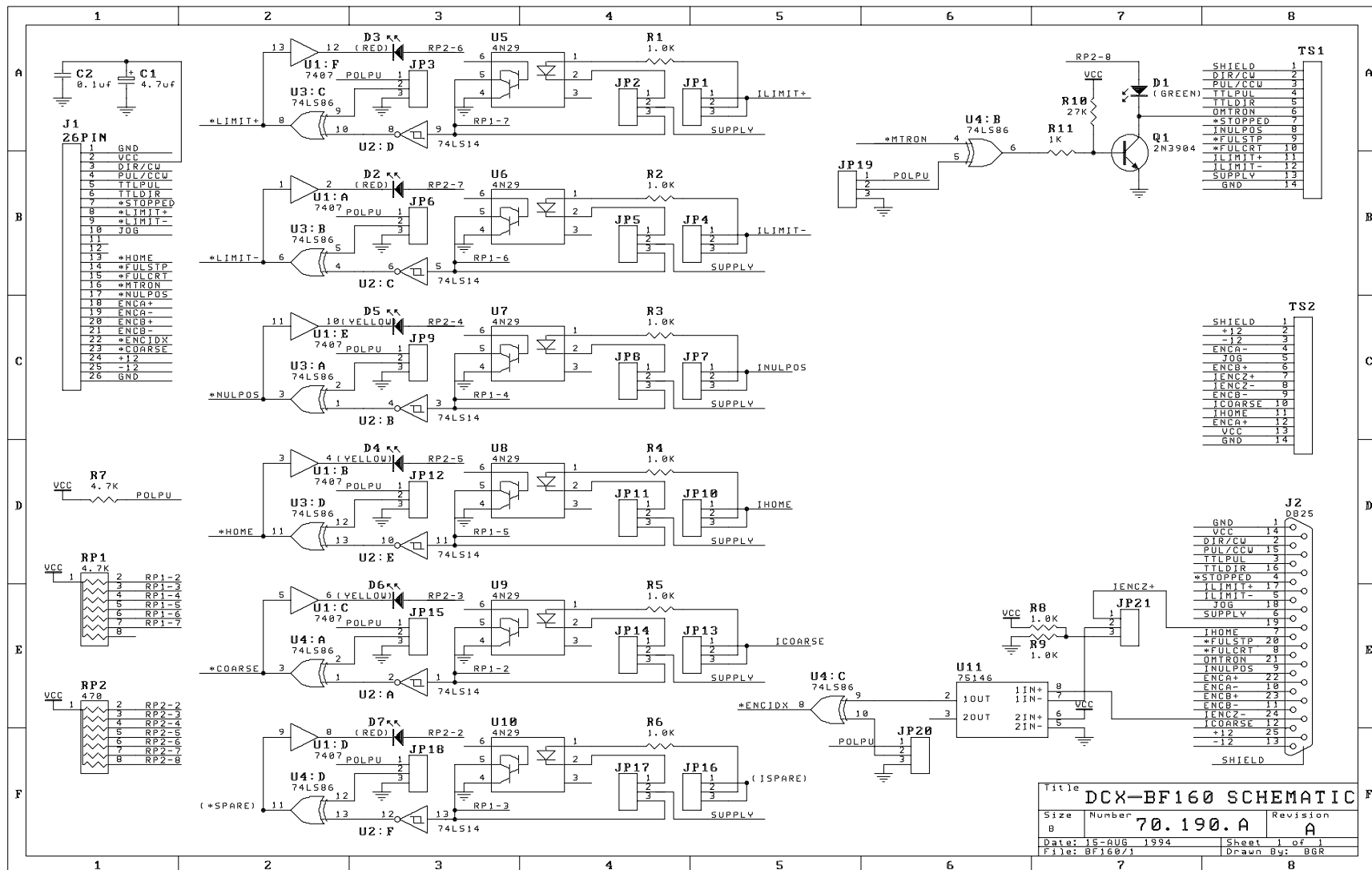
<b>Pin</b>	<b>Description</b>
1	Shield
2	Direction/CW Pulse
3	Pulse/CCW Pulse
4	Reserved
5	Reserved
6	Motor On
7	Stopped
8	Null Position
9	Full/Half Step
10	Full/Half current
11	Limit +
12	Limit -
13	Opto Supply
14	Ground

**Terminal strip TS2**

<b>Pin</b>	<b>Description</b>
1	Shield
2	+12 VDC
3	-12 VDC
4	Aux. Encoder Phase A-
5	Jog
6	Aux. Encoder Phase B+
7	Aux. Encoder Index+
8	Aux. Encoder Index-
9	Aux. Encoder Phase B-
10	Aux. Enc. Coarse Home
11	Home
12	Aux. Encoder Phase A+
13	+5 VDC
14	Ground

DCX-BF160 Interface layout









## Chapter Contents

---

- Introduction to MCCL (low level command set)
- MCCL Command Quick Reference Tables
- Building MCCL Macro Sequences
- MCCL Multi-Tasking
- Downloading MCCL Text Files
- Single Stepping MCCL Programs
- Outputting Formatted Messages Strings
- PLC I/O Control using MCCL Sequence Commands
- PLC Control and DCX Analog I/O
- DCX User Registers
- Reading Data from DCX Memory
- DCX Scratch Pad Memory
- MCCL Command Set Description

## DCX MCCL Commands

---

### 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** (Move Absolute).

Any MCCL command that references an axis is preceded by an axis specifier *a* (*aMA*). To issue a move absolute to axis #1:

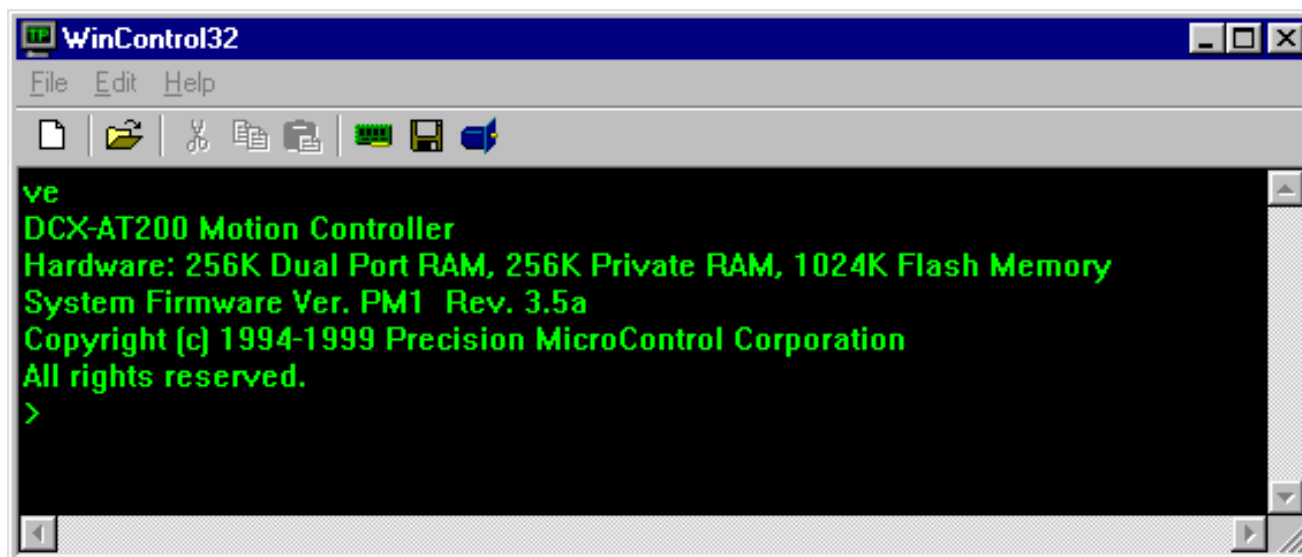
**1MA** (axis #1 Move Absolute)

Most DCX commands will also include a parameter value following the two character mnemonic. This parameter is identified as *n* (*aMA<sub>n</sub>*). To move axis #1 to absolute position 10.25:

**1MA10.25** (axis #1 Move Absolute 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:**  $aMRn$   $a$  = Axis number  $n$  = integer or real  $\geq 0$

**compatibility:** MC200, MC210, MC260

**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
AG	E2h	set Acceleration feed-forward Gain
AH	EAh	Auxiliary encoder define Home
BD	DOh	Backlash compensation Distance
DB	76h	set position DeadBand
DG	E3h	set Deceleration feed-forward Gain
DH	23h	Define Home
DI	24h	Direction
DS	75h	Deceleration Set
DT	C5h	Delay at Target
FC	40h	Full Current
FF	33h	amplifier Fault input off
FN	32h	amplifier Fault input oN
FR	27h	set derivative sampling period
HC	41h	Half Current
HL	D3h	set motion High Limit
HS	E8h	set High Speed
IL	28h	set Integration Limit
JA	38h	Jog Acceleration
JB	DFh	Jog deadBand
JG	DCh	Jog proportional Gain
JO	DEh	Jog Offset
JV	37h	Jog Velocity
LF	36h	motion Limits off
LL	D2h	set motion Low Limit
LM	34h	Limit Mode
LN	35h	motion Limits oN
LS	36h	set Low Speed
MS	E7h	set Medium Speed
MV	C4h	set Minimum Velocity
OO	CBh	set the Output Offset
OM	D8h	set Output Mode
PH	73h	set servo output PHase
SA	2Bh	Set Acceleration
SD	2Ch	Set Derivative gain
SE	19h	Stop on Error
SF	3Eh	Step Full
SG	2Dh	Set prop. Gain of motor
SH	3Fh	Step Half
SI	2Eh	Set Integral gain
SQ	74h	Set TorQue
SS		Set Slave ratio
SV	2Fh	Set Velocity
UK	D7h	set User output constant
UA	9Ch	Use as default Axis
UO	B3h	set User Offset
UR	B1h	set User Rate conversion
UP	9Dh	Use Physical axis
US	AFh	set User Scale
UT	B2h	set User Time conversion
UZ	B0h	set User Zero
VA	ADh	set Vector Acceleration
VD	AEh	set Vector Deceleration
VG	77h	set Velocity Gain
VO	E0h	set Velocity Override
VV	ACH	set Vector Velocity

## Mode Commands

MCCL	Code	Description
CM	1Bh	enable Contour Mode (arcs and lines)
GM	8h	enable Gain Mode (no velocity profile)
IM	72h	Input Mode (closed loop stepper)
PM	17h	enable Position Mode
SM		enable Master/Slave mode
TQ	9H	enable TorQue mode
VM	18h	enable Velocity Mode

## Motion Commands

MCCL	Code	Description
AB	Ah	ABort
AF	EBh	Auxiliary encoder arm/Find index
BF	CFh	Backlash compensation off
BN	CEh	Backlash compensation oN
CA	B4h	arc Center Absolute
CD	C1h	Contour Distance
CP	C0h	Contour Path
CR	B5h	arc Center Relative
EA		arc Ending Angle absolute
FE	Bh	Find Edge
FI	Ch	Find Index
GH	Dh	Go Home
GO	Eh	GO
HO	Fh	HOme
IA	5Dh	Index Arm
JF	3Ah	Jogging off
JN	39h	Jogging oN
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
NS	ABh	No Synchronization
PP	EDh	Profile Parabolic
PR		Recode motion data
PS	EEh	Profile S-curve
PT	EFh	Profile Trapezoidal
RC	115h	Restore Configuration
RR		aRc Radius
SC	114h	Save Configuration
SN	AAh	Synchronization oN
ST	16h	STop

## 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
GA	F8h	Get Analog value
GD		Get module ID
GU	89h	Get the default axis
GX	F7h	Get auXiliary encoder position
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 into accumulator
SL	90h	Shift Left accumulator n bits
SR	91h	Shift Right accumulator n bits
TR	57h	Tell contents of Register n
WB	99h	Write accumulator Byte to n
WD	95h	Write accumulator double to n
WL	9Bh	Write accumulator Long to n
WV	94h	Write accumulator float to n
WW	9Ah	Write accumulator Word to n

## 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

## Reporting Commands

MCCL	Code	Description
AT	E9h	Auxiliary encoder Tell position
AZ	ECh	Auxiliary encoder tell index
DO		Display recorded optimal position
DQ		Display recorded DAC output
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 position Gain
TI	4Fh	Tell Integral gain
TK	5Ch	Tell velocity constant
TL	50h	Tell integration Limit
TM	51h	Tell stored Macros
TO	59h	Tell Optimal
TP	52h	Tell Position
TR	57h	Tell Register n
TQ	D1h	Tell torQue
TS	53h	Tell Status
TT	54h	Tell Target
TV	55h	Tell Velocity
TX	58h	Tell contouring count
TZ	5Ah	Tell index position
VE	56h	tell VErSion

## 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 Stop

## I/O Commands

<b>MCCL</b>	<b>Code</b>	<b>Description</b>
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
TA	49h	Tell the value of Analog input
TC	4Ah	Tell state of digital Channel

## Miscellaneous Commands

<b>MCCL</b>	<b>Code</b>	<b>Description</b>
BR	1Eh	Baud Rate
DM	3Ch	Decimal Mode
EF	25h	Echo off
EN	26h	Echo oN
FM	10Dh	Free Memory
HE	48h	HElp
HF	30h	Handshake off
HM	3Dh	Hexadecimal Mode
HN	31h	Handshake oN
ME	10Ch	MEmory allocate
NO	78h	No Operation
PC	80h	set Prompt Character
RT	2Ah	ReseT system
XF	7Eh	Xon/Xoff Protocol oFF
XN	7Dh	Xon/Xoff Protocol oN

## File Commands

<b>MCCL</b>	<b>Code</b>	<b>Description</b>
DL	10Fh	Directory Listing
FO	10Eh	FOrmat file system
LO	112h	LOad file
RF	111h	Remove File
TY	110h	TYpe file

## Plotting Commands

<b>MCCL</b>	<b>Code</b>	<b>Description</b>
PA	121h	Plotter Acceleration
PD	125h	Pen Down macro
PE	119h	Plotting Enable
PF	118h	Plot File
PI	123h	Plotter Initialize macro
PQ	122h	Plotter Quick velocity
PU	124h	Pen Up macro
PV	120h	Plotter Velocity
PX	11Ah	Plotter X axis
PY	11Bh	Plotter Y axis
SP	126h	Select Pen macro
XO	11Eh	plotter X Offset
XS	11Ch	plotter X Scale
YO	11Fh	plotter Y Offset
YS	11Dh	plotter Y Scale

## Building MCCL Macro Sequences

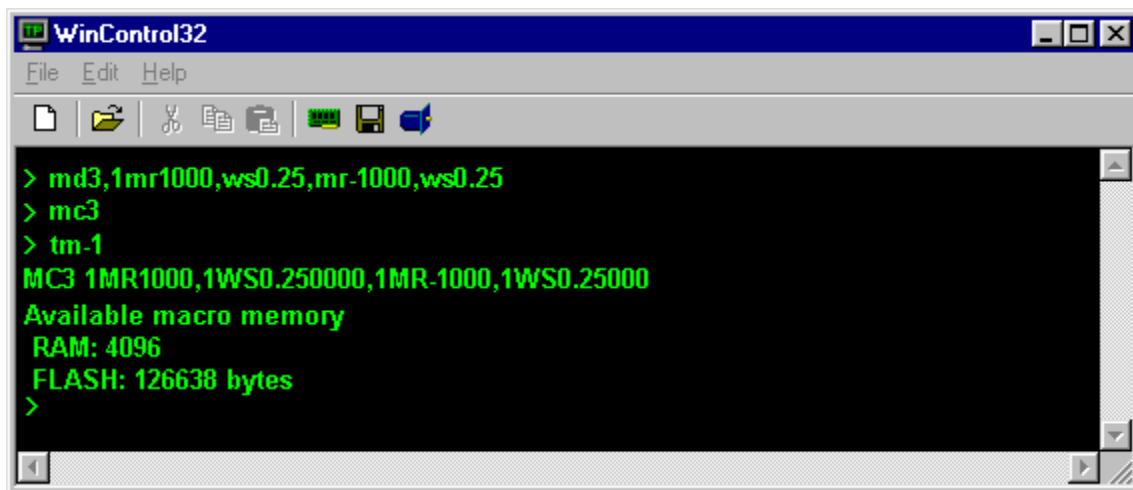
A powerful feature of the DCX is the ability to define MCCL sequences as macros. This simply means giving a short name to a sequence of commands to form a new command defined by the user. For example:

```
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 Tell Macro (TMn) 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**.

The DCX can store up to 1100 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).

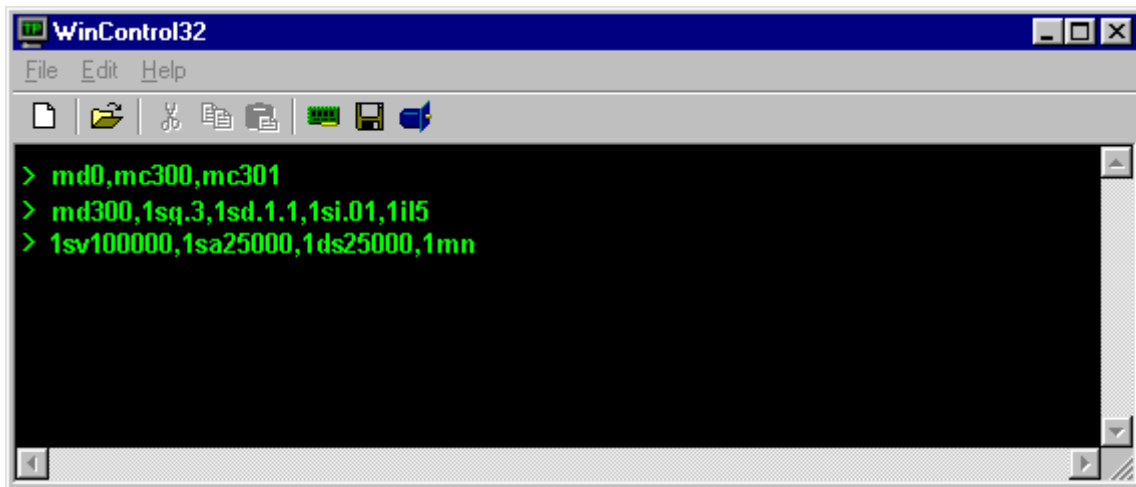
Macro numbers 0 through 9, and 256 through 1099 are stored in the on-board Flash memory. These macros will be retained when power to the board is turned off. However, once a macro that is stored in Flash memory is defined, it can't be redefined, only undefined by the **Reset Stored macros** (RM)



command. Macro numbers 10 through 255 are stored in on-board RAM memory. These macros will not be retained when the power is off. The macros in RAM can be redefined or undefined as long as memory is available. The Reset Macro (RMn) command can be used to erase the macros stored in RAM memory, in Flash memory, or in both, by using the appropriate command parameter.

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 Tell Macro (TMn) command can be used to display the amount of Flash and RAM memory available for macros storage at any give time.

Another feature of the DCX is that macro 0 will be executed on power up or reset. A common use for this feature is to have the motors automatically configured when the board is powered up. To accomplish this: define a set of macros that contain commands to set the motor parameters. These macros should be located in flash memory so they are preserved when the power is turned off. Now define macro 0 to call these setup macros. Each time the DCX is powered up or reset, macro 0 will execute which will configure the motors. The graphic below shows using macro 0 to call macro's 300 (define PID parameters) and 301 (define trajectory parameters and turn on the motor).



**Warning:** If macro 0 contains a command sequence that runs indefinitely, the command interface will be busy while the macro is executing. This will cause **all MCAPI programs and utilities to be unable to communicate with the DCX**. If this happens and MCAPI is unable to communicate with the DCX:



- 1) Turn off power to the computer
- 2) Remove the DCX, set the memory offset rotary switch (SW1) to position F
- 3) Install the DCX in the PC, turn on power for 10 seconds
- 4) Turn off power to the computer
- 5) Reset the memory offset rotary switch (SW1) to its original position
- 6) Reinstall the DCX into the computer, turn on the power
- 7) All macro's should have been deleted. Communication with MCAPI should be restored.

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 DCX to execute continuous monitoring or control sequences while still communicating 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 Generate Task (GTn) 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.



**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.

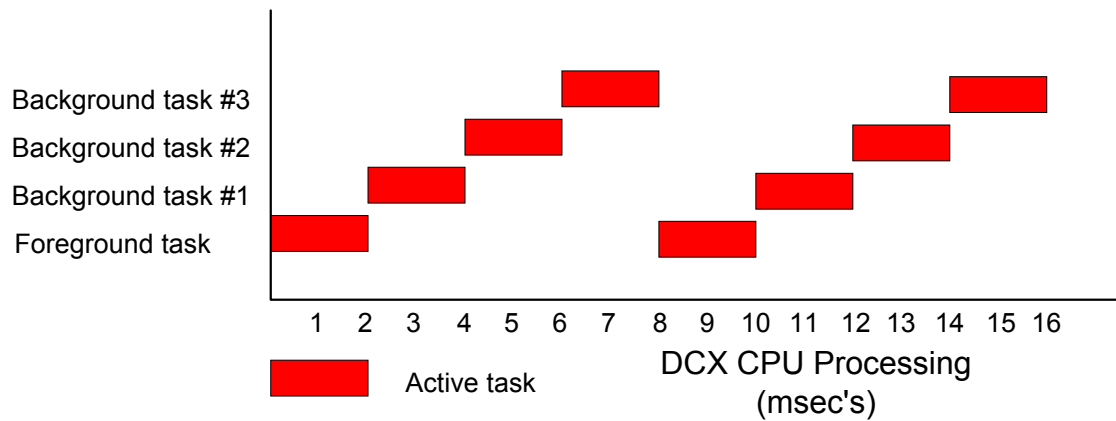
Another way to create a background task is to place the Generate Task command as the first command in a command line, using a parameter of 0. This instructs the command interpreter to take all the commands that follow the Generate Task command and cause them to run as a background task. The commands will run identically to commands placed in a macro and generated as a task.

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 2 msec's. 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 2 msec's of processor time every 8 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 Break command is executed. When a task is terminated, the resources it required are made available to run other background tasks. Alternatively, the Escape Task command can be used to force a background task to terminate. 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.

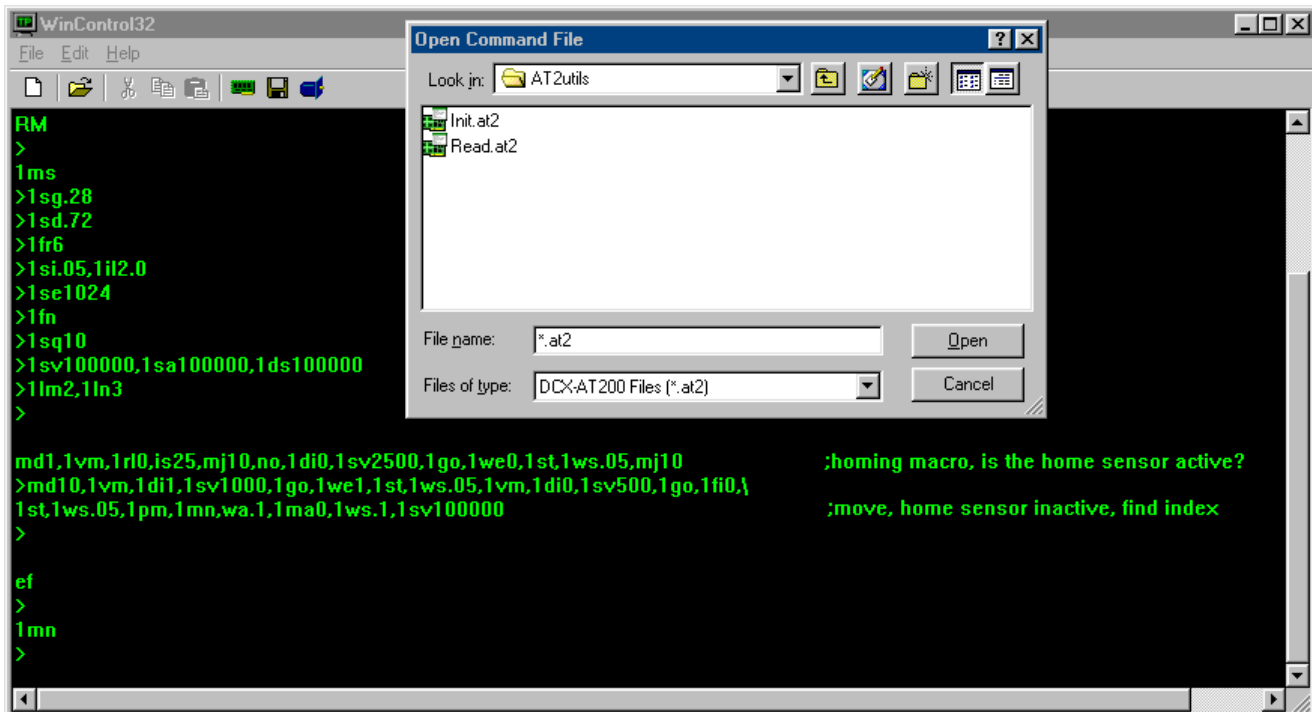
As an example, the graphic below depicts a background task where the action of axis one will depend on the state of three digital inputs. If digital input #5 is on, axis one will move 1.5 inches in the positive direction and then dwell for 100 msec's. If digital input #6 is on, axis one will move 1.5 inches in the negative direction and then dwell for 100 msec's. In foreground, a command loop will monitor the state of digital input 2, if at anytime this channel goes true, the background task will be terminated.

## 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.

## 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
```

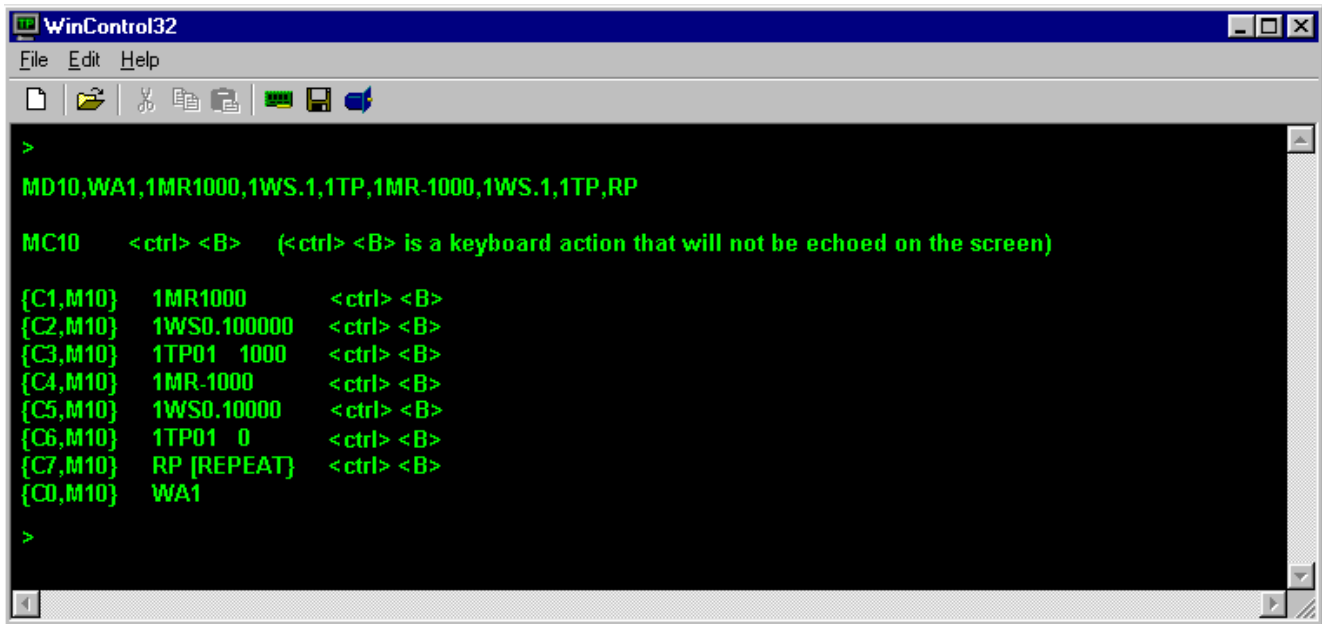
This sample program will: wait for 1 second, move 1000 encoder counts, report the position 100 msec's after the calculated trajectory is complete, move -1000 encoder counts, report the position 100 msec's after the calculated trajectory is complete, repeat the command sequence.

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



```
WinControl32
File Edit Help
[Icons]
>
MD10,WA1,1MR1000,1WS.1,1TP,1MR-1000,1WS.1,1TP,RP
MC10 <ctrl> <B> (<ctrl> <B> is a keyboard action that will not be echoed on the screen)
{C1,M10} 1MR1000 <ctrl> <B>
{C2,M10} 1WS0.100000 <ctrl> <B>
{C3,M10} 1TP01 1000 <ctrl> <B>
{C4,M10} 1MR-1000 <ctrl> <B>
{C5,M10} 1WS0.100000 <ctrl> <B>
{C6,M10} 1TP01 0 <ctrl> <B>
{C7,M10} RP [REPEAT] <ctrl> <B>
{C8,M10} WA1
>
```

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.



Note: Firmware revision 3.5b or higher is required for single step mode

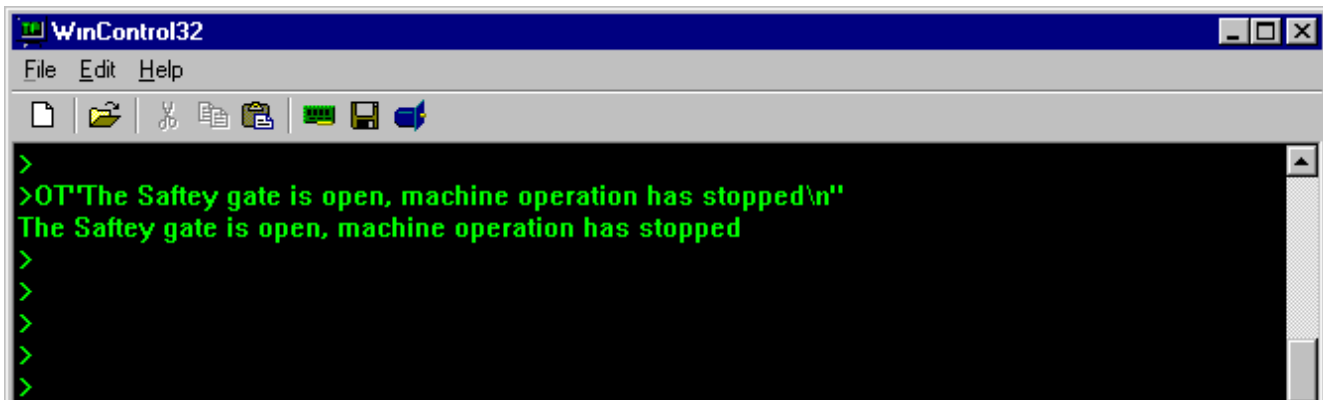
## Outputting Formatted Message Strings

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
```

## PLC I/O Control using MCCL Sequence Commands

### PLC control and DCX Digital I/O

The following graphic depicts a fluid dispensing system. A liquid adhesive is applied to a gasket. Two linear axes are mounted to the table top (see the slides with bellows) and are slaved together to move the dispensing head back and forth. The fluid dispensing head is moved left and right by a third axis.

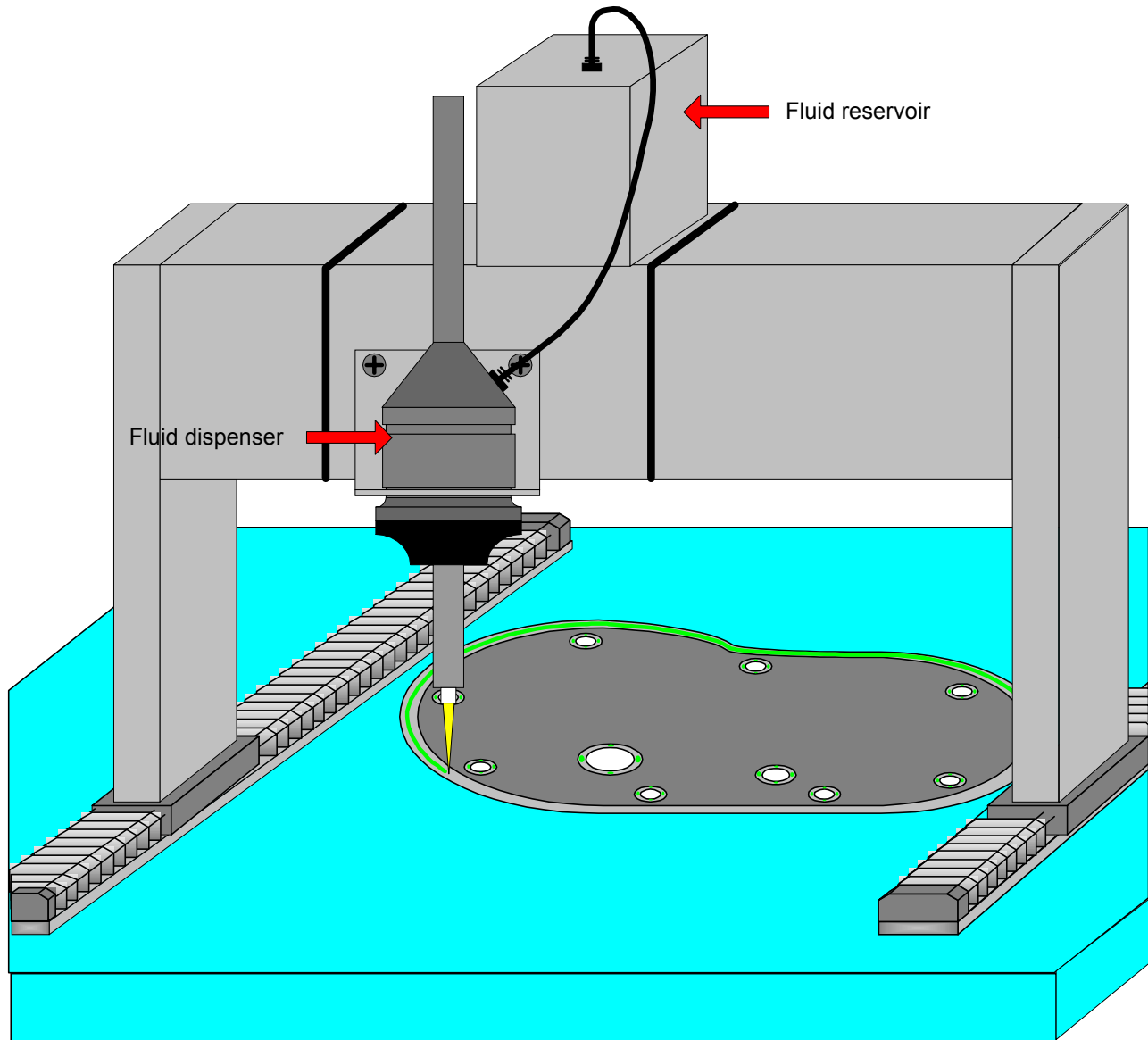
For this application there is no Z (up and down) axis motion, the operator manually positions the dispense head for the proper height. The following Digital I/O are used dispensing the liquid:

**Inputs:**

Fluid reservoir empty (dig. I/O #1)  
 Dispense valve empty (dig. I/O #2)  
 Valve busy being primed (dig. I/O #3)

**Outputs:**

Turn on the dispense valve air supply (dig. I/O #4)  
 Dispense liquid – turn on Archimedes motor (dig I/O #5)  
 Stop dispense – reverse Archimedes motor (dig I/O #6)  
 Prime the valve (dig I/O #7)



A 'PC based' application program is used by the operator to initialize and operate the dispensing system. The 'Fluid Reservoir' and 'Dispense Valve' sensors are monitored by MCCL macro's executing as background tasks. If either of these sensors 'goes active', a DCX User Register flag will be set, The application program will then notify the operator to remedy the error condition. The following macro sequence will monitor the state of the two sensors:

```
MD300,IF1,MJ301,NO,IF2,MJ302,NO,WA.1,JR-7      ;pole sensors for error condition
MD301,1VO0,AL1,AR201,OT"The fluid reservoir level is low, add fluid and prime
the dispense valve \n"                          ;stop the motion (Velocity
                                                ;Override =0), set Input Sensor Error
                                                ;Flag, output ASCII error message
MD302,1VO0,AL2,AR201,OT"The dispense valve fluid level is low, add fluid and
prime the dispense valve \n"                    ;stop the motion (Velocity
                                                ;Override =0), set Input Sensor Error
                                                ;flag, output ASCII error message

GT300,AR200                                     ;execute sensor monitoring macros as a
                                                ;background task. Load the task ID in
                                                ;register #200.

AL0,AR201,1VO100,GT300,AR200                   ;after error condition cleared, resume
                                                ;operation
```

Prior to initiating a dispensing operation:

- Move the axes to the starting position (handled by the "PC based" application program)
- Verify that no error conditions exist
- Begin fluid output operation
- Begin motion (handled by the "PC based" application program)

A sequence of DCX macro's control the dispensing of fluid.

```
MD400,AL@201,IE1,MJ401,NO,IE2,NO,MJ402,MJ403    ;check for input sensors active (error
                                                ;condition)
MD401,OT"The fluid reservoir level is low /n"
MD402,OT"The dispense valve fluid is low /n"

MD403,CN4,WA.150,CN5                            ;begin fluid dispense

MD404,CF5,CF6,WA.150,CF4                        ;terminate fluid dispense
```

The following commands are used to program conditional (If/Then) command execution based on the state of a digital channel:

### Do if channel oFf

**MCCL command:**   DFx x = Channel number

Used for conditional execution of commands. If the specified digital I/O channel 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
```

### Do if channel 'x' is oN

**MCCL command:**   DNx x = Channel number



Used for conditional execution of commands. If the specified digital I/O channel 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
```

### If channel oFf do next command, else skip 2 commands

**MCCL command:** IFx x= Channel number

Used for conditional execution of commands. If the specified digital I/O channel 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
```

### If channel ' oN do next command, else skip 2 commands

**MCCL command:** INx x= Channel number

Used for conditional execution of commands. If the specified digital I/O channel 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
```

### Wait for digital channel oFf

**MCCL command:** WFxx = 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.

### Wait for digital channel oN

**MCCL command:** WNxx = Channel number

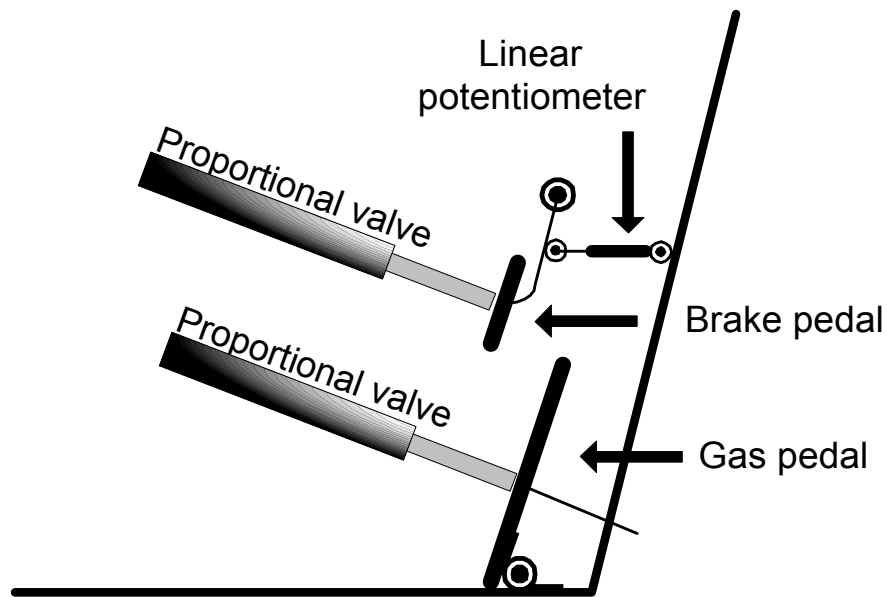
**compatibility:** MC400

**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.

## PLC Control and DCX Analog I/O

Remote operation of a automobile provides a simple example analog I/O control. Two Proportional Pneumatic valves are used to control the velocity and the braking. One valve is positioned to depress the 'gas pedal' to control the vehicle speed. The other is positioned to depress the 'brake pedal'. An analog tachometer is connected to the drive train to provide the vehicle speed feedback. A linear potentiometer is mounted to the back side of the brake pedal to provide the feedback of the position/force of the braking action.



### Vehicle Speed Control

An analog output of 0.0V will cause the proportional valve that depresses the gas pedal to be fully retracted (no velocity). An analog output of +5.0V causes the valve to fully extend (pedal to the metal). The user defines a 'look up table' that is used to equate a DAC value to the speed of the vehicle. The user's application program then writes the appropriate DAC (speed) value into DCX User Register #100 and sets the new speed command flag (register 102). For the purposes of this example, if the difference between the commanded speed and the actual speed of the vehicle is greater than 5%, the DAC output will be adjusted accordingly. The following DCX User Registers are used to store and manipulate data for this application:

User Register 100	;current speed command
User Register 101	;A/D conversion of the output of the tachometer
User Register 102	;new programmed 'speed command' flag
User Register 103	;difference between the speed command and the tachometer ;feedback (signed value)
User Register 104	;difference between the speed command and the tachometer ;feedback (unsigned value)
User Register 105	;+/- speed command tolerance (5%)

The following macro commands will output the user's 'speed command' and adjust the vehicle velocity:

```

MD10,AL@100,OA1,MJ11           ;output the DAC value
MD11,WA.10,GA5,AR101,MJ12       ;wait 100 msec's, load the output of
                                ;the tachometer
MD12,AL@100,AS@101,AR103,AE3,AR104,MJ13 ;find the signed and unsigned value of
                                ;the difference between the 'speed
                                ;command' and the tachometer feedback
MD13,AL@100,AM.05,AR105,MJ14     ;calculate 5% of the 'speed command'
MD14,AL@104,IB@105,MJ10,NO,MJ15 ;is vehicle speed within 5% of
                                ;commanded speed?
MD15,AL@103,IB0,MJ16,NO,IG0,MJ17 ;is vehicle velocity too fast or too
                                ;slow?
MD16,AL@100,AS@104,AL@100,AL0,AR102,MJ10 ;decrease 'speed command ' by 5%
MD17,AL@100,AA@104,AL@100,AL0,AR102,MJ10 ;increase 'speed command ' by 5%

GT10,AR200                       ;execute the 'speed control' macro
                                ;sequence as a background task, store
                                ;Task ID# in user register 200

```

### Braking Control

An analog output of 0.0V will cause the proportional valve that depresses the gas pedal to be fully retracted (no brake pedal pressure). An analog output of +5.0V causes the valve to fully extend (full brake pressure). The user defines a 'look up table' that is used to equate a DAC value to the brake pedal pressure. The user's application program then writes the appropriate DAC (brake) value into DCX User Register #111 and sets the braking flag (register 101). For the purposes of this example, if the difference between the commanded brake pressure and the brake pedal position feedback is greater than 5%, the DAC output will be adjusted accordingly. The following DCX User Registers are used to store and manipulate data for this application:

User Register 110	; "Braking" flag register, 1=braking
User Register 111	; current DAC braking command
User Register 113	; Brake pedal linear potentiometer feedback
User Register 114	; difference between the brake command and the linear potentiometer ; feedback (signed value)
User Register 115	; difference between the brake command and the linear potentiometer ; feedback (unsigned value)
User Register 116	; +/- brake command tolerance (20%, 10%, or 5%)

The following macro commands will output the user's 'brake command' and adjust the brake pedal position:

```

MD20,AL@110,IE1,MJ21,NO,WA.1,RP ;braking flag set?
MD21,AL@111,OA2,MJ22             ;output DAC 'brake command'
MD22,WA.1,GA6,AR113,MJ23         ;wait 100 msec's, load the A/D linear
                                ;potentiometer value
MD23,AL@111,AS113,AR114,AE3,AR115,MJ24 ;find the signed and unsigned value of
                                ;the difference between the 'brake
                                ;command' and the brake pedal
                                ;potentiometer feedback
MD24,AL@111,AM.05,AR116,MJ25     ;calculate 5% of the 'speed command'
MD25,AL@115,IB@116,MJ32,NO,MJ26 ;is brake pedal within 5% of command
                                ;position?
MD26,AL@114,IB0,MJ28,NO,IG0,MJ27 ;is brake pedal pressure too much or
                                ;too little?

```

---

```

MD27,AL@111,AS@115,AL@110,MJ20      ;decrease 'brake pedal pressure' by 5%
MD28,AL@111,AA@1115,AL@110,MJ20      ;increase 'brake pedal pressure' by 5%

GT20,AR201                             ;execute the 'braking' macro sequence
                                       ;as a background task. Store the Task
                                       ;ID# in user register 201

```

## 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
AA <sub>n</sub>	Accumulator Add (ACC = ACC + n)
AC <sub>n</sub>	Accumulator Complement, bit wise (ACC = !ACC)
AD <sub>n</sub>	Accumulator Divide (ACC = ACC/n)
AE <sub>n</sub>	Accumulator logical Exclusive or with n, bit wise (ACC = ACC eor n)
AL <sub>n</sub>	Accumulator Load with constant n (ACC = n)
AM <sub>n</sub>	Accumulator Multiply(ACC = ACC x n)
AN <sub>n</sub>	Accumulator logical aNd with n, bit wise (ACC = ACC and n)
AO <sub>n</sub>	Accumulator logical Or with n, bit wise (ACC = ACC or n)
AR <sub>n</sub>	copy Accumulator to Register n (REG <sub>n</sub> = ACC <sub>n</sub> )
AS <sub>n</sub>	Accumulator Subtract (ACC = ACC - n)
GAX	Get Analog value (ACC = channel x)
aGX	Get auXiliary encoder position (ACC = axis a auxiliary encoder)
IB <sub>n</sub>	If accumulator is Below (>) n, do next command, else skip 2 commands
IC <sub>n</sub>	If bit n of accumulator is Clear, do next command, else skip 2 commands
IE <sub>n</sub>	If accumulator Equals constant n, do next command, else skip 2 commands
IG <sub>n</sub>	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)

## Reading Data from DCX Memory

A group of read commands are available for accessing the DCX's internal memory. These commands provide an easy method of moving motor data in and out of the Accumulator (user register 0). To use a read command for this purpose, it should include an axis specifier 'a' and a parameter 'n' selected from the motor table offsets listed below. 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.

Examples of using the read commands to access the motor tables are shown below.

To load the status of axis 2 into the accumulator, issue the following command:

```
2RL0          ;load the status of axis #2 into the accumulator
```

To load the position of axis 3 the accumulator, issue the following command:

```
3RD20        ;load the position of axis #3 into the accumulator
```

Memory read/write commands:

aRBn Read Byte (8 bit) at memory location n into accumulator (ACC = (n))  
aRDn Read Double at memory location n into accumulator (ACC = (n))  
aRLn Read Long (32 bit) at memory location n into accumulator (ACC = (n))  
aRVn Read float at memory location n into accumulator (ACC = (n))  
aRWn Read Word (16 bit) at memory location n into accumulator (ACC = (n))

Motor Table Entries – 32 bit integer (long)

<b>Motor Table Entry Description</b>	<b>Offset (decimal)</b>
Motor Status	0
Position Count	4
Optimal Count	8
Index Count	12
Auxiliary Status	16
Module Base Address	252

Motor Table Entries – 64 bit floating point (double)

<b>Motor Table Entry Description</b>	<b>Offset (decimal)</b>
Position	20

Target	28
Optimal Position	36
Breakpoint Position	44
Position Dead band	52
Maximum Following Error	60
Soft Motion Limit Setting (low)	68
Soft Motion Limit Setting (high)	76
User Scale	84
User Zero	92
User Offset	100
User Rate Conversion	108
User Output Constant	116
Programmed Velocity	124
Programmed Acceleration	132
Programmed Deceleration	140
Minimum Velocity	148
Minimum Velocity	156
Jog Acceleration	220
Jog Minimum Velocity	228

Motor Table Entries – 32 bit floating point (float)

<b>Motor Table Entry Description</b>	<b>Offset (decimal)</b>
Velocity Gain	164
Acceleration Gain	168
Deceleration Gain	172
Velocity Override	176
Torque Limit	180
Proportional Gain	184
Derivative Gain	188
Integral Gain	192
Integration Limit	196
Module Analog Input 1	200
Module Analog Input 2	204
Jog Gain	208
Jog Offset	212
Jog Dead band	216

Motor Table Entries – 16 bit integer (word)

<b>Motor Table Entry Description</b>	<b>Offset (decimal)</b>
Wait Stop Timer	236
Wait Target Timer	238
Sampling Frequency	240
Master Axis	242
Module Status	244
Axis Number	246
Module Position	248

Module Type	250
-------------	-----

## DCX Scratch Pad Memory

Over and above what is available by using the User Registers, the DCX also provides an 8KB space allocated for user scratch pad memory. The allocate Memory (ME<sub>n</sub>) command is used to format the memory space for user operations. The parameter *n* defines the number of bytes to be allocated for use.

Upon executing the memory allocate command, the accumulator will be loaded with the address of the first byte of allocated memory. The following commands are used to write data from the accumulator into the allocated memory locations:

WB<sub>n</sub> Write accumulator low Byte (8 bit) to memory location *n* ((*n*) = ACC)  
WD<sub>n</sub> Write accumulator Double to absolute memory location *n* ((*n*) = ACC)  
WL<sub>n</sub> Write accumulator Long (32 bit) to memory location *n* ((*n*) = ACC)  
WV<sub>n</sub> Write accumulator float to absolute memory location *n* ((*n*) = ACC)  
WW<sub>n</sub> Write accumulator low Word (16 bit) to memory location *n* ((*n*) = ACC)

The following commands are used to read data from the allocated memory into the accumulator:

RB<sub>n</sub> Read Byte (8 bit) at memory location *n* into accumulator (ACC = (*n*))  
RD<sub>n</sub> Read Double at memory location *n* into accumulator (ACC = (*n*))  
RL<sub>n</sub> Read Long (32 bit) at memory location *n* into accumulator (ACC = (*n*))  
RV<sub>n</sub> Read float at memory location *n* into accumulator (ACC = (*n*))  
RW<sub>n</sub> Read Word (16 bit) at memory location *n* into accumulator (ACC = (*n*))

The Free Memory (FM<sub>n</sub>) command returns previously allocated memory and returns it to the 'heap' from which it was allocated. The parameter *n* of this command must be the same as the value that was loaded into the accumulator upon issuing the memory allocated (ME) command.



# MCCL Command Set Description

## Setup Commands

### Acceleration Gain

**MCCL command:** `aAGn`  $a$  = Axis number  $n$  = integer or real  $\geq 0$

**compatibility:** MC200, MC210

**see also:** DG, VG

This command sets the acceleration feed-forward gain for a servo. The product of this gain and the motor's calculated acceleration will be summed into the controller's DAC output. The acceleration gain is only applied while a motor is accelerating, when it decelerates the deceleration gain term is used (see DG command).

**comment:** Acceleration and deceleration feed-forwards are not calculated when a motor is in contour mode.

```
1AG10.0                                ;Sets Acceleration gain for axis 1 to
                                         ;10.0
```

### Auxiliary encoder define Home

**MCCL command:** `aAHn`  $a$  = Axis number  $n$  = integer or real  $\geq 0$

**compatibility:** MC200, MC210, MC260

**see also:** DH

This command causes axis  $a$  auxiliary encoder position to be set to  $n$ . This encoder input is available on both the MC200 and MC260 modules, and is used for loop closure when a MC260 is controlling a closed loop stepper. The auxiliary encoder of a MC200 is used for position verification only, it cannot be used for dual loop positioning. For defining the home position of the primary encoder, see the Define Home command.

### Backlash compensation Distance

**MCCL command:** `aBDn`  $a$  = Axis number  $n$  = integer or real  $\geq 0$

**compatibility:** MC200, MC210, MC260

**see also:** BF, BN

Use this command to set the distance required to nullify the effects of mechanical backlash in the system. The command parameter should be equal to half the amount the motor must move to take up backlash when it changes direction. The units for this command parameter are encoder counts, or the units established by the User Scale command for the axis.

Once the backlash compensation distance is set, issuing the Backlash compensation oN command will cause the controller to add or subtract the distance from the motor's commanded position during all subsequent moves. If the motor moves in a positive direction, the distance will be added; if the motor moves in a negative direction, it will be subtracted. When the motor finishes a move, it will remain in the compensated position until the next move. See the description on backlash compensation in the **Application Solutions** chapter of this manual.

## set position DeadBand

**MCCL command:** aDBn    a = Axis number    n = integer or real >= 0

**compatibility:** MC200, MC210

**see also:** DT

This command sets the position dead band that is used by the controller to determine when a servo axis is 'At Target'. In order for the At Target flag in the motor status to be set, a servo must remain within the specified dead band of the current target position for a period of time specified with the Delay at Target (aDTn) command.

## Define Home

**MCCL command:** aDHn    a = Axis number    n = integer or real >= 0

**compatibility:** MC200, MC210, MC260

**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.

## Deceleration Set

**MCCL command:** aDSn    a = Axis number    n = integer or real >= 0

**compatibility:** MC200, MC210, MC260

**see also:** SA, SV

Defines the deceleration rate for an axis. The default units for the command parameter are encoder counts (or steps) per second per second.

## Direction

**MCCL command:** aDIn    a = Axis number    n = integer or real >= 0

**compatibility:** MC200, MC210, MC260

**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.

## amplifier Fault oFf

**MCCL command:** aFFn    a = Axis number    n = none

**compatibility:** MC200

**see also:** FF

Disables the Amplifier Fault input of a servo module. See description of amplifier Fault input oN command (FN), for further details.

## amplifier Fault oN

**MCCL command :** aFNn    a = Axis number    n = none

**compatibility:** MC200

**see also:** FF

Enables the Amplifier Fault input of a servo module. If the input goes active after this command is executed, the motor will be turned off 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. This command has no effect on stepper motors.

## Deceleration Gain

**MCCL command:** aDGn a = Axis number n = integer or real >= 0

**compatibility:** MC200, MC210

**see also:** AG, VG

This command sets the deceleration feed-forward gain for a servo. The product of this gain and the motor's calculated deceleration will be summed into the controller's DAC output. The deceleration gain is only applied while a motor is decelerating. When it accelerates the acceleration gain term is used (see AG command).

**comment:** Acceleration and deceleration feed-forwards are not calculated when a motor is in contour mode.

```
example: 1DG10.0                ;Sets Deceleration gain for axis 1 to
                                   ;10.0
```

## Delay at Target

**MCCL command:** aDTn a = Axis number n = integer or real >= 0

**compatibility:** MC200, MC210

**see also:** DB

This command sets the time period during which a servo must remain within the position dead band of the target for the 'At Target' flag in the motor status to be set.

## set the derivative sampling period

**MCCL command:** aFRn a = Axis number n = integer >= 0

**compatibility:** MC200, MC210

**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 zero. For a value of n, the sampling period will be (n + 1) \* (sample period). See the High Speed command for a discussion of the sample period on servos. See **Tuning the Servo** section in the **Motion Control** chapter.

## Full Current

**MCCL command :** aFC a = Axis number

**compatibility:** MC260

**see also:** HC

Causes Full/Half Current output signal of a stepper module to go low.

## Jog Acceleration

**MCCL command:** aJAn a = Axis number n = integer or real >= 0

**compatibility:** MC200, MC210, MC260

**see also:** JN, JG

Sets jogging acceleration. See the description of **Jogging** in the **Motion Control** chapter.

## Jog deadBand

**MCCL command:** aJBn a = Axis number n = integer or real >= 0

**compatibility:** MC200, MC210, MC260

**see also:** JN, JF

Sets jogging joystick input dead band. See the description of **Jogging** in the **Motion Control** chapter.

## Jog proportional Gain

**MCCL command:** aJGn    a = Axis number    n = integer or real >= 0

**compatibility:** MC200, MC210, MC260

**see also:** JF, JN

Sets jogging joystick proportional gain. See the description of **Jogging** in the **Motion Control** chapter.

## Jog Offset

**MCCL command:** aJOn    a = Axis number    n = integer or real >= 0

**compatibility:** MC200, MC210, MC260

**see also:** JF, JN

Sets jogging joystick input offset. See the description of **Jogging** in the **Motion Control** chapter.

## Jog minimum Velocity

**MCCL command:** aJVn    a = Axis number    n = integer or real >= 0

**compatibility:** MC260

**see also:** JN, JG

Sets jogging minimum velocity for stepper motors. See the description of **Jogging** in the **Motion Control** chapter.

## Half Current

**MCCL command:** aHC    a = Axis number

**compatibility:** MC260

**see also:** FC

Causes Full/Half Current output signal of a stepper module to go high.

## High motion soft Limit

**MCCL command:** aHLn    a = Axis number    n = integer or real

**compatibility:** MC200, MC210, MC260

**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.

**comment:** When the axis is in contouring mode, this limit will be tripped anytime the desired or true position is greater than the limit regardless of commanded direction. Thus, to move an axis out of the limit region, it must be placed in a non-contour mode. When one or more axes are moving in contour

mode, and one of the axes experiences a limit trip, all axes associated with the motion will be turned off or stopped.

## High speed

**MCCL command:** aHSn a = Axis number n = none

**compatibility:** MC200, MC210, MC260

**see also:** LS, MS

This command has a different effect depending on whether it is issued to a servo or stepper motor axis. For a servo axis, it sets the servo feedback loop to 4 KHz update rate (**with no integral term**). For a stepper motor axis, it sets the maximum pulse rate to 1.25 Million Pulses/Sec.

## Integration Limit

**MCCL command:** aLIn a = Axis number n = integer or real >= 0

**compatibility:** MC200, MC210

**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.

## motion Limits oFf

**MCCL command:** aLFn a = Axis number n = 0 – 15

**compatibility:** MC200, MC210, MC260

**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.

## Low motion soft Limit

**MCCL command:** aLLn a = Axis number n = integer or real

**compatibility:** MC200, MC210, MC260

**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.

**comment:** When the axis is in contouring mode, this limit will be tripped anytime the desired or true position is less than the limit regardless of commanded direction. Thus, to move an axis out of the limit region, it must be placed in a non-contour mode. When one or more axes are moving in contour mode, and one of the axes experiences a limit trip, all axes associated with the motion will be turned off or stopped.

## Limit Mode

**MCCL command:** aLMn    a = Axis number    n = integer 0 - 15

**compatibility:** MC200, MC210, MC260

**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 on an axis. The command parameter should be formed by adding a value of 0, 1, or 2 for the hard limit switch mode, to a value of 0, 4 or 8 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. 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.

## Limits oN

**MCCL command:** aLNn    a = Axis number    n = 0 - 15

**compatibility:** MC200, MC210, MC260

**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. If a soft motion limit is enabled, and the respective axis goes 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. 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.

## Low Speed

**MCCL command:** aLSn    a = Axis number    n = none

**compatibility:** MC200, MC210, MC260

**see also:** HS, MS

This command has a different effect depending on whether it is issued to a servo or stepper motor axis. For a servo axis, it sets the feedback loop to 1 KHz update rate. For a stepper motor axis, it sets the maximum pulse rate to 19.5 Thousand Pulses/Sec.

## Medium Speed

**MCCL command:** aMSn    a = Axis number    n = none

**compatibility:** MC200, MC210, MC260

**see also:** HS, LS

This command has a different effect depending on whether it is issued to a servo or stepper motor axis. For a servo axis, it sets the feedback loop to 2 KHz update rate. For a stepper motor axis, it sets the maximum pulse rate to 156 Thousand Pulses/Sec.

## set Minimum Velocity

**MCCL command:** aMVn    a = Axis number    n = integer or real >= 0

**compatibility:** MC260

**see also:** SV

Sets the minimum velocity for a given stepper motor axis. The purpose of this command is to set an initial and final velocity for motion of stepper motors. Below this velocity a full stepping motor is 'cogging' between steps. The default units for the command parameter are steps per second. This command will have no effect on servos.

## Output Dead band

**MCCL command:** aODn    a = Axis number    n = integer or real > 0, <=10  
**compatibility:** MC200, MC210  
**see also:** OO

This command can be used to simulate a 'frictionless servo system'. The value *n* defines a voltage dead band range in the output of a MC200 and MC210 servo module. Parameter *n* modifies the commanded analog (MC200) or motor drive (MC210) output to a servo. The value *n* is added to a positive output and subtracted from a negative output.

## Output Mode

**MCCL command:** aOMn    a = Axis number    n = integer 0, 1, 2, or 3  
**compatibility:** MC200, MC210, MC260  
**see also:**

This command is used to set a servo or stepper module's output mode. The available modes are listed in the following tables.

<i>n</i>	<b>MC200 Output Mode</b>
0	Bipolar Analog output, -10V to +10V
1	Unipolar Analog output, 0V to +10V, direction J3 pin 7
2	Bipolar PWM signal output on J3 pin 7, 0 - 50% duty cycle
3	Unipolar PWM signal output on J3 pin 7, 0 – 100% duty cycle, Direction on Analog Output (J3 pin 2)

<i>n</i>	<b>MC260 Output Mode</b>
0	Pulse and Direction outputs (default)
1	CW and CCW Pulse Outputs

## Output Offset

**MCCL command:** aOOn    a = Axis number    n = integer or real >= -10, <= +10  
**compatibility:** MC200, MC210  
**see also:** OD

This command is used to provide software programmability of the zero point of a servo output. Similar to adjusting an offset potentiometer, the parameter *n* will redefine the 'no commanded motion' output level.

## set the servo output PHasing

**MCCL command:** aPHn    a = Axis number    n = 0 or 1  
**compatibility:** MC200, MC210  
**see also:** OM

This command is used to set a servo module's output phasing. The phase of the output will determine whether the module drives the servo in a direction that reduces position error, or increases it. The

module defaults to standard phasing, which is the same as issuing this command with a parameter of 0. The module output can be set to reverse phase by issuing this command with a parameter of 1.

## Set Acceleration

**MCCL command:** aSAn a = Axis number n = integer or real >= 0

**compatibility:** MC200, MC210, MC260

**see also:** DS, SV

Set the maximum acceleration rate for a given axis. The default units for the command parameter are encoder counts (or steps) per second per second.

## Set Derivative gain

**MCCL command:** aSDn a = Axis number n = integer or real >= 0

**compatibility:** MC200, MC210

**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.

## Stop on following Error

**MCCL command :** aSEn a = Axis number n = integer or real, 0 to disable

**compatibility:** MC200, MC210

**see also:**

Used to set the maximum following or position error for a servo. 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. Issuing this command with a parameter of 0 will disable errors on excessive following errors.

## Step Full

**MCCL command:** aSF a = Axis number n = none

**compatibility:** MC260

**see also:** SH

Causes Full/Half Step output signal of a stepper module to go low. This command is typically used to disable 'micro stepping' of a stepper driver.

## Step Half

**MCCL command:** aSH a = Axis number n = none

**compatibility:** MC260

**see also:** SF

Causes Full/Half Step output signal of stepper module to go high. This command is typically used to enable 'micro stepping' of a stepper driver.

## Set the Integral gain

**MCCL command :** aSIn a = Axis number n = integer or real >= 0

**compatibility:** MC200, MC210



**see also:** 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.

## set Proportional gain

**MCCL command:** aSGn a = Axis number n = integer or real >= 0.000153, <=10

**compatibility:** MC200, MC210

**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 stiffening the force holding a servo in position. The parameter to this command has default units of volts per encoder count. This command should not be used for open loop stepper axes. See the description of **Tuning the Servo** in the **Motion Control** chapter.

## Set torQue

**MCCL command:** aSQn a = Axis number n = integer or real >= -10, <= 10

**compatibility:** MC200, MC210

**see also:** QM, PM, VM

Sets maximum output level for servos. When an axis is placed in torque mode, this command sets the continuous output level. The default units for the command parameter are volts. See the description of **Torque Mode Output Control** in the **Applications Solutions** chapter.

## Set the ratio of Slave

**MCCL command:** aSSn a = Axis number n = integer or real > 0

**compatibility:** MC200, MC210

**see also:** SM

This command specifies the ratio at which the slave axis (designated by *a*) will move relative to a changed in encoder counts (or steps) of the master axis. As soon as the Set Master command is issued, the slave axis will begin tracking the master axis with the programmed ratio. The controller makes the position calculations using the optimal positions of the master and slave axes when the Set Master command was issued as the starting point. See the description of **Master/Slave motion** in the **Motion Control** chapter.

## Set Velocity

**MCCL command:** aSVn a = Axis number n = integer or real >= 0

**compatibility:** MC200, MC210, MC260

**see also:** SA, DS

Set the maximum velocity for a given axis. The default units for the command parameter are encoder counts (or steps) per second.

## set the defaUlt Axis

**MCCL command:** aUAN a = Axis number n = integer > 0, < 6

**compatibility:** MC200, MC210, MC260

**see also:**

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*.

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

## User Konstant

**MCCL command:** aUK*n*    *a* = Axis number    *n* = integer or real >= 0

**compatibility:** MC200, MC210

**see also:**

This command is used to configure an axis for commands in user units. The default setting is 1.0. See the description of **Setting User Units** in the **Application Solutions** chapter.

## User Offset

**MCCL command:** aUOn    *a* = Axis number    *n* = integer or real >= 0

**compatibility:** MC200, MC210, MC260

**see also:**

This command is used to configure an axis for commands in user units. The default setting is 1.0. See the description of **Setting User Units** in the **Application Solutions** chapter.

## Use Physical axis addressing

**MCCL command:** aUP*n*    *a* = Axis number    *n* = integer > 0, < 6

**compatibility:** MC200, MC210, MC260

**see also:**

This command is used to reassign the axis designator of a motor module. The value *a* should equal the new axis designator. The parameter *n* should equal the current physical location of the motor module. See the description of **Physical Assignment of Axes Numbers** in the **Appendix**.

## User Rate

**MCCL command:** aUR*n*    *a* = Axis number    *n* = integer or real >= 0

**compatibility:** MC200, MC210, MC260

**see also:**

This command is used to configure an axis for commands in user units. The default setting is 1.0. See the description of **Setting User Units** in the **Application Solutions** chapter.

## User Scale

**MCCL command :** aUS*n*    *a* = Axis number    *n* = integer or real >= 0

**compatibility:** MC200, MC210, MC260

**see also:**

This command is used to configure an axis for commands in user units. The default setting is 1.0. See the description of **Setting User Units** in the **Application Solutions** chapter.

## User Time

**MCCL command:** `aUTn`  $a$  = Axis number  $n$  = integer or real  $\geq 0$

**compatibility:** MC200, MC210, MC260

**see also:**

This command is used to configure an axis for commands in user units. The default setting is 1.0. See the description of **Setting User Units** in the **Application Solutions** chapter.

## set the User zero position

**MCCL command:** `aUZn`  $a$  = Axis number  $n$  = integer or real  $\geq 0$

**compatibility:** MC200, MC210, MC260

**see also:**

This command is used to configure an axis for commands in user units. The default setting is 1.0. See the description of **Setting User Units** in the **Application Solutions** chapter.

## Vector Acceleration

**MCCL command:** `aVAN`  $a$  = Axis number  $n$  = integer or real  $\geq 0$

**compatibility:** MC200, MC210, MC260

**see also:** CP, VD, VV

This command specifies the acceleration rate for motion along a contour path. It should be issued to the controlling axis prior to the first Contour Path command. It can also be issued to the controlling axis while motion is in progress, but it will take effect immediately, and be used for all succeeding motion. See the description of **Contour Motion (lines and arcs)** in the **Motion Control** chapter.

## Vector Deceleration

**MCCL command:** `aVDn`  $a$  = Axis number  $n$  = integer or real  $\geq 0$

**compatibility:** MC200, MC210, MC260

**see also:** CP, VA, VV

This command specifies the deceleration rate for motion along a contour path. It should be issued to the controlling axis prior to the first Contour Path command. It can also be issued to the controlling axis while motion is in progress, but it will take effect immediately, and be used for all succeeding motion. See the description of **Contour Motion (lines and arcs)** in the **Motion Control** chapter.

## Velocity Override

**MCCL command:** `aVOn`  $a$  = Axis number  $n$  = integer or real  $\geq 0$

**compatibility:** MC200, MC210, MC260

**see also:** CP, VV

Sets a multiplying factor that will be applied to the velocity of both servo and stepper motors. This command does not support jogging. For contour moves (linear, circular) the axis identified 'a' should be the axis number of the 'controlling' axis. See the description of **Contour Motion (lines and arcs)** in the **Motion Control** chapter.

## Vector Velocity

**MCCL command:** aVVn    a = Axis number    n = integer or real >= 0

**compatibility:** MC200, MC210, MC260

**see also:** CP, VA, VD

This command specifies the maximum velocity for motion along a contour path. It should be issued to the controlling axis prior to the first Contour Path command. When a Contour Path command is issued, the current vector velocity will be stored with the move in the motion table. The Vector Velocity command can also be issued to the controlling axis while motion is in progress, but it won't have any effect on the contour path motions already issued. To adjust the velocity of motions already in progress, use the Velocity Override command. See the description of **Contour Motion (lines and arcs)** in the **Motion Control chapter**.

## Velocity gain

**MCCL command :** aVGn    a = Axis number    n = integer or real >= 0

**compatibility:** MC200, MC210, MC260

**see also:** AG, DG

Sets the feed forward gain of the servo PID-FF loop. The default units for the parameter to this command are volts per encoder counts per second. For example, if the Velocity gain of a servo is set to 0.0001, the feed forward component of the modules output will be 1 volt at a speed of 10000 encoder counts per second ( $0.0001 * 10000 = 1$ ). The parameter to this command can be a positive or negative number. This command should not be used for open loop stepper motors.

## Mode Commands

### Contour Mode

**MCCL command:** aCMn a = Axis number n = integer > 0, <6

**compatibility:** MC200, MC210, MC260

**see also:** CP

This command places a servo or stepper motor in the Contour Mode of operation. The parameter to this command specifies the controlling axis for a group of axes to be included in contour path commands. The controlling axis should be the lowest numbered axis in the group. Contour Mode is terminated by issuing a Position (aPM) or Velocity Mode (aVMn) command to all axes in the group. The controlling axis should be taken out of contour mode last. See the description of **Contour Motion (lines and arcs)** in the **Motion Control** chapter.

### enable Gain mode

**MCCL command:** aGMn a = Axis number n = none

**compatibility:** MC200, MC210

**see also:** IL, SD, SG, SI

This command places a servo in the Gain Mode of operation. In this mode, the servo can be commanded to execute moves to specific positions. However, no velocity profile (maximum velocity, acceleration, or deceleration) will be calculated. The servo will be driven to the new target based **only upon** the output of the PID loop.

### Input Mode

**MCCL command:** aIMn a = Axis number n = 0 or 1

**compatibility:** MC260

**see also:**

The Input Mode command issued with a parameter of 1 enables closed loop stepper motion. Issued with n = 0 disables closed loop motion. See the description of **DCX Stepper Basics** in the **Motion Control** chapter.

### Master/Slave mode

**MCCL command:** aSMn a = Axis number n = integer > 0, <= 101

**compatibility:** MC200, MC210, MC260

**see also:** SS

This command will cause axis 'a' to be "slaved" to a "master" axis n with a ratio specified by the Set Slave ratio command. Alternatively, this command can slave one axis to two master axes for tangential knife control in cutter applications. In this case the command parameter is determined by the following algorithm:

$$\text{parameter} = \text{master 1 axis number} + (\text{master 2 axis number} \times 16)$$

Issuing this command with a parameter of zero to the slave axis, will terminate the connection to the master axis. See the Master/Slave Motion section of this manual for further details. See the description of **Master/Slave Motion** and **Tangential Knife Control** in the **Motion Control** chapter.

## Position Mode

**MCCL command:** aGMn a = Axis number n = none

**compatibility:** MC200, MC210, MC260

**see also:** MA, MR

This command places a servo or stepper motor 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, parabolic or S-curve velocity profile. When in the Position Mode with trapezoidal velocity profile selected, servos can change the move destination while the move is in progress. With stepper motors, the destination can be changed as long as it doesn't require the motors direction to change. If it is necessary to change the direction of a stepper motor, it must first be stopped and a new move command issued. 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.

## torQue Mode

**MCCL command:** aQMn a = Axis number n = none

**compatibility:** MC200, MC210

**see also:** SQ

This command places a servo (not valid for steppers) in the Torque Mode of operation. This command does **not imply** that the torque generated by or current across the motor is monitored or controlled by the DCX controller. In this mode, the output drive signal to the motor will hold a constant level as specified with Set TorQue command. The parameter to this command has default units of volts. In this mode of operation, the servo motor control module (MC200, MC210) is 'turned into' programmable power supply. As such, the change of position as indicated by the encoder of an axis, while still recorded by the servo module, will have no affect on the operation of the controller. See the description of **Torque Mode Output Control** in the **Applications Solutions** chapter.

## Velocity Mode

**MCCL command:** aVMn a = Axis number n = none

**compatibility:** MC200, MC210, MC260

**see also:** DI

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.

## Motion Commands

### ABort motion

**MCCL command:** aAB a = Axis number n = none

**compatibility:** MC200, MC210, MC260

**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. For a stepper motor, the pulses from the module will be disabled immediately. For both servos and stepper motors, the target position of the axis is set equal to the present 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
```

### Auxiliary encoder find index mark

**MCCL command:** aAFn a = Axis number n = integer or real >= 0

**compatibility:** MC200, MC210, MC260

**see also:** AH

This command is used to initialize a motor's auxiliary encoder at a given position. It will remain in effect until the auxiliary encoder's index pulse goes active. At that time the current position of the auxiliary encoder will be set to n. See the description of **Homing Axes** in the **Motion Control** chapter.

### Backlash compensation oFf

**MCCL command:** aBFn a = Axis number n = integer or real >= 0

**compatibility:** MC200, MC210

**see also:** BD, BN

Use this command to disable backlash compensation. As soon as this command is executed, the motor will move to its uncompensated position. See the description of **Backlash Compensation** in the **Application Solutions** chapter.

### Backlash compensation oN

**MCCL command:** aBN a = Axis number n = none

**compatibility:** MC200, MC210

**see also:** BD, BF

Use this command to enable backlash compensation. It should be issued after the backlash compensation distance has set been with the BD command. Prior to issuing the Backlash Compensation On command, the motor should be positioned halfway between the two positions where it makes contact with the mechanical gearing. This will allow the controller to take up the backlash (when the first move in either direction is made) without 'bumping' the mechanical position.

While backlash compensation is enabled, the response to the Tell Position, Tell Target and Tell Optimal commands will be adjusted to reflect the ideal positions (as if no mechanical backlash were present). See the description of **Backlash Compensation** in the **Application Solutions** chapter.

### arc Center Absolute

**MCCL command:** aCA $n$   $a$  = Axis number  $n$  = integer or real  $\geq 0$

**compatibility:** MC200, MC210, MC260

**see also:** CM, CP

This command is used to specify the center of an arc for a Contour Path motion. Since the arc motion is performed by two axes, this command (or the arc Center Relative command) should occur twice in a Contour Path command that initiates the arc motion. The parameter to this command specifies the center of the arc for the selected axis in absolute user units. See the description of **Contour Motion** in the **Motion Control** chapter.

### arc Center Relative

**MCCL command:** aCR $n$   $a$  = Axis number  $n$  = integer or real  $\geq 0$

**compatibility:** MC200, MC210, MC260

**see also:** CM, CP

This command is used to specify the center of an arc for a Contour Path motion. Since the arc motion is performed by two axes, this command (or the arc Center Absolute command) should occur twice in a Contour Path command that initiates the arc motion. The parameter to this command specifies the center of the arc for the selected axis in user units, relative to its' target position prior to beginning the arc motion. See the description of **Contour Motion** in the **Motion Control** chapter.

### arc Ending Angle absolute

**MCCL command:** aEA $n$   $a$  = Axis number  $n$  = integer or real  $\geq 0$

**compatibility:** MC200, MC210, MC260

**see also:** CM, CP

This command is used to specify the ending angle (end point) of a contour arc move. The parameter  $n$  is expressed as an absolute angle relative to when the axes were last homed. This command would be used in conjunction with the Center Absolute, Center Relative, or aRc Radius commands. See the description of **Contour Motion** in the **Motion Control** chapter.

### arc Ending Angle relative

**MCCL command:** aER $n$   $a$  = Axis number  $n$  = integer or real  $\geq 0$

**compatibility:** MC200, MC210, MC260

**see also:** CM, CP

This command is used to specify the ending angle (end point) of a contour arc move. The parameter  $n$  is expressed as an angle relative to when the axes were last homed. This command would be used in conjunction with the Center Absolute, Center Relative, or aRc Radius commands. See the description of **Contour Motion** in the **Motion Control** chapter.

### Radius of aRc

**MCCL command:** aRR $n$   $a$  = Axis number  $n$  = integer or real  $\geq 0$

**compatibility:** MC200, MC210, MC260

**see also:** CM, CP

This command specifies the radius of a contour mode arc. For an arc of less than 180 degrees the parameter  $n$  should be a positive value equal to the radius of the arc. For an arc of greater than 180



degrees the parameter  $n$  should be a negative value equal to the radius of the arc. See the description of **Contour Motion** in the **Motion Control** chapter.

## Contour Distance

**MCCL command:**  $aCDn$   $a$  = Axis number of the controlling axis  $n$  = integer or real  $\geq 0$

**compatibility:** MC200, MC210, MC260

**see also:** CM, CP

For user defined contour moves, this command is used to specify the distance as measured along the path from the contour path starting point to the end of the next motion. For typical orthogonal (X, Y, Z) geometry, the DCX calculates the contour move distance ( $\sqrt{X^2+Y^2+Z^2}$ ) based on the target positions specified by the move absolute and/or move relative commands. Parameter  $n$  of the Contour Distance command allows the user to enter a custom contour distance to be used for trajectory generation. The value  $n$  is used as an ending point for the contour path motion and to determine the proper velocity over the motion segment. See the description of **Contour Motion** in the **Motion Control** chapter.

## define the Contour Path

**MCCL command:**  $aCPn$   $a$  = Axis number  $n$  = integer 0, 1, 2, or 3

**compatibility:** MC200, MC210, MC260

**see also:** CM

This command is used to form a 'compound' command that specifies a multi axis motion. The compound command must begin with the Contour Path command, followed by a variable number of other motion commands. The axis number used with the Contour Path command must be the controlling axis in a group of motors that have been previously placed in Contour Mode. The parameter to the Contour Path command selects either a linear, arc or 'user defined' motion. The table below list the type of motion each parameter value specifies, and the acceptable commands that can be include in the compound command. See the description of **Contour Motion** in the **Motion Control** chapter.

```
1CP1,1GH,2GH,3GH,1VV60000
1CP1,1MA10000,2MA20000,3MR-5000,1VV30000
1CP1,1GH,2GH,3GH
1CP2,1CA20000,2CA0,1MA40000,2MA0
1CP3,1CR-20000,2CR0,1MR-40000,2MR0
```

Parameter $n$	Motion Type	Compatible Commands
0	User defined	CD,GH,MA,MR,VV
1	Linear	GH,MA,MR,VV
2	Clockwise arc	CA,CR,GH,MA,MR,VV
3	Counter-Clockwise arc	CA,CR,GH,MA,MR,VV

## Find Edge

**MCCL command:**  $aFEn$   $a$  = Axis number  $n$  = integer or real  $\geq 0$

**compatibility:** MC260

**see also:** FI

This command is used to initialize a stepper motor at a given position. The command will remain in effect until the home input of the module goes active. At that time an internal position register of the MC260 module will be set to the current position. The position of the axis (where the index mark was captured) will not be defined to position  $n$  until after the Motor Off / Motor on (aMNn) command sequence has been issued. This command will not cause any motor motion to be started or stopped.

It is up to the user to initiate motor motion before issuing the command, and to stop any motion after it completes. See the description of **Homing Axes** in the **Motion Control** chapter.

**Note:** The status bit associated with the Find Edge command is bit 24 (stepper Home). The Find Edge command causes this bit to be latched after the index mark has been captured. To clear the latched status bit, issue the Motor Off and Motor on sequence.

```
MD1,1LM2,1LN3,MJ10           ;call homing macro
MD10,1VM,1DI0,1SV10000,1GO,1RL0,IS24,MJ11,NO,IS17,MJ13,NO,JR-7
                                ;test for sensors (home and +limit)
MD11,1ST,1WS.1,1DI0,1SV5000,1GO,1RL0,IC24,MJ12,NO,JR-4
                                ;Move positive until home sensor off
MD12,1ST,1WS.1,1DI1,1SV5000,1GO,MJ15
                                ;move back to the home sensor
MD13,1MN,1DI1,1SV5000,1GO,MJ15 ;move out of limit sensor range back
                                ;toward the home sensor
MD14,1FE0,1ST,1WS.1,1MF,WA.1,1MN,1PM,1MA0
                                ;find the active edge of the home
                                ;sensor. Stop axis, initialize
                                ;position, move to position 0.
```

## Find Index

**MCCL command:** aFIn a = Axis number n = integer or real >= 0

**compatibility:** MC200, MC210

**see also:** DH, FE

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. At that time the current position of the servo will be set to *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,1RL0,IS25,MJ11,NO,IS17,MJ12,NO,JR-7
                                ;test for sensors (home and +limit)
MD11,1ST,1WS.01,1DI1,1GO,1WE1,1ST,1DI0,1GO,1WE0,1FI0,1ST,1WS.01,1PM,1MN,1MA0
                                ;if home sensor true, initialize on
                                ;index
MD12,1MN,1DI1,1GO,1WE0,MJ11   ;move negative until home true
```

See the description of **Homing Axes** in the **Motion Control** chapter.

## Go Home

**MCCL command:** aGH a = Axis number n = none

**compatibility:** MC200, MC210, MC260

**see also:** MA, MC, MD

Causes the specified axis or axes to move to the offset position that was specified when the last DH, FI or FE command was issued. This is equivalent to a Move Absolute command, where the destination is 0 or the offset of the home position.

## GO

**MCCL command:** aGOn a = Axis number n = integer 0 or 1

**compatibility:** MC200, MC210, MC260

**see also:** CM, VM

Causes one or all axes to begin motion in velocity or contour mode. In contour mode, synchronization must be on. The parameter to this command is only used for contour mode, and determines whether the motions will be linearly interpolated ( $n = 0$ ), or a cubic spline ( $n = 1$ ).

## HOMe

**MCCL command:** aHO  $a = \text{Axis number}$   $n = \text{none}$

**compatibility:** MC200, MC210, MC260

**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.

## Index Arm

**MCCL command:** aAn  $a = \text{Axis number}$   $n = \text{integer or real } \geq 0$

**compatibility:** MC200, MC210

**see also:** FI, WI

This command is used to arm the index capture function of a servo axis. It has a similar function to Find Index, but does not wait for the index pulse to occur. After the Index Arm command is issued, and the index pulse occurs, the location where the index pulse occurred will be defined captured. After the index pulse has occurred (indicated by status bit 24 = 1), the Wait for Index and Motor oN commands are issued to define the location of the index pulse as position  $n$ .

## Jogging oFf

**MCCL command:** aJF  $a = \text{Axis number}$   $n = \text{none}$

**compatibility:** MC200, MC210, MC260

**see also:** JN

Disables jogging of a servo or stepper motor. See the description of **Jogging** in the **Motion Control** chapter.

## Jogging oN

**MCCL command:** aJN  $a = \text{Axis number}$   $n = \text{none}$

**compatibility:** MC200, MC210, MC260

**see also:** JF

Enables jogging of a servo or stepper motor. See the description of **Jogging** in the **Motion Control** chapter.

## Learn Position

**MCCL command:** aLPn  $a = \text{Axis number}$   $n = \text{integer } \geq 0, \leq 1536$

**compatibility:** MC200, MC210, MC260

**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.

## Learn Target

**MCCL command:** aLT*n*    *a* = Axis number    *n* = integer >= 0, <=1536

**compatibility:** MC200, MC210, MC260

**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.

## Motor oFf

**MCCL command:** aMF    *a* = Axis number    *n* = none

**compatibility:** MC200, MC210, MC260

**see also:** MN

Issuing this command will place one or all servos and stepper motors 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. For stepper motors, the Motor On output will go inactive. This command can be used to prevent unwanted motion or to allow manual positioning of the servo or stepper motor.

## Motor oN

**MCCL command:** aMN    *a* = Axis number    *n* = none

**compatibility:** MC200, MC210, MC260

**see also:** MF

Use this command to place one or all servos and stepper 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 or a stepper motor module's drive enable output signal will go active. This has the effect of causing servo and stepper motors to hold their 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.

## Move to Point

**MCCL command:** aMP $n$   $a$  = Axis number  $n$  = integer  $\geq 0$ ,  $\leq 1536$

**compatibility:** MC200, MC210, MC260

**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.

## Move Relative

**MCCL command:** aMR $n$   $a$  = Axis number  $n$  = integer or real  $\geq 0$

**compatibility:** MC200, MC210, MC260

**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.

## Parabolic Profile

**MCCL command:** aPP  $a$  = Axis number  $n$  = none

**compatibility:** MC260

**see also:** PS, PT

This command causes the respective stepper motor to perform **point to point** motions with a triangular acceleration profile. The resulting velocity profile is parabolic. Motion with this profile is limited to **position and contour** mode moves, where the acceleration, deceleration, , velocity, and destination **don't change during the move**. See the description of **Defining the Characteristics of a Move** in the **Motion Control** chapter.

## Record axis data

**MCCL command:** aPR $n$   $a$  = Axis number  $n$  = integer  $> 0$ ,  $\leq 512$

**compatibility:** MC200, MC210

**see also:**

This command is used to begin the recording of motion data (actual position, optimal position, and DAC output) for an axis. See the description of **Record and display Motion Data** in the **Application Solutions** chapter.

## Profile S-curve

**MCCL command:** aPS  $a$  = Axis number  $n$  = none

**compatibility:** MC200, MC210

**see also:** PP, PT

This command causes the respective servo motor to perform **point to point** motions with a sinusoidal acceleration profile. The resulting velocity profile is trapezoidal with rounded corners, thus the name S-curve. Motion with this profile is limited to **position and contour** mode moves, where the

acceleration, deceleration, velocity, and destination **don't change during the move**. See the description of **Defining the Characteristics of a Move** in the **Motion Control** chapter.

## Profile Trapezoidal

**MCCL command:** aPT     a = Axis number     n = none

**compatibility:** MC200, MC210, MC260

**see also:** PS, PT

This command causes the respective servo or stepper motor to perform point to point motions with a constant acceleration profile. The resulting velocity profile is trapezoidal. When motion is being performed with this profile, the acceleration, velocity, and destination can be changed at any time during the move. See the description of **Defining the Characteristics of a Move** in the **Motion Control** chapter.

## Restore Configuration

**MCCL command:** aRCn     n = Axis number     n = integer > 0, <=127

**compatibility:** MC200, MC210, MC260

**see also:** SC

This command takes axis specifier *a* and requires a file number as parameter *n*. This command restores the entire motor table. This includes the public motor table in dual port memory and the private motor table in internal RAM. When used with the Save Configuration command, motors can be stopped (aVO0) during a move, their configurations saved, switched to any other mode (except contouring), moved about and then returned to their original positions, their configurations restored, and then commanded to continue the contour move (aVO1.0). See the description of **Pause and Resume Motion** in the **Motion Control** chapter.

## Save Configuration

**MCCL command:** aSCn     n = Axis number     n = integer > 0, <=127

**compatibility:** MC200, MC210, MC260

**see also:** RC

This command takes axis specifier *a* and requires a file number parameter *n*. This command saves the entire motor table. This includes the public motor table in dual port memory and the private motor table in internal RAM. When used with the Restore Configuration command, motors can be stopped (aVO0) during a move, their configurations saved, switched to any other mode (except contouring), moved about and then returned to their original positions, their configurations restored, and then commanded to continue the contour move (aVO1.0). See the description of **Pause and Resume Motion** in the **Motion Control** chapter.

## Stop

**MCCL command:** aST     a = Axis number

**compatibility:** MC200, MC210, MC260

**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.

## No Synchronization

**MCCL command:** aNS      a = Axis number  
**compatibility:** MC200, MC210, MC260  
**see also:** SN

This command turns synchronization off in contour path motions. It should be issued to the controlling axis of the contour group. See the description of **Contour Motion** in the **Motion Control** chapter.

## Synchronization oN

**MCCL command:** aSN      a = Axis number  
**compatibility:** MC200, MC210, MC260  
**see also:** GO, CP, NS

This command turns synchronization on for contour path motion. This command can be issued to the controlling axis prior to Contour Path commands. With synchronization on, no motion will occur when a Contour Path command is issued, until a succeeding GO command is issued to the controlling axis. See the description of **Contour Motion** in the **Motion Control** chapter.

## 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.

### Auxiliary encoder Tell position

**MCCL command:** aAT*p*    *a* = Axis number    *p* = 0, .1, .2, .3, .4, .5

**compatibility:** MC200, MC210, MC260

**see also:** AH

Reports the absolute position of the auxiliary encoder of an axis. To read the primary encoder or stepper position, see the Tell Position command.

### Auxiliary encoder tell index

**MCCL command:** aAZ*p*    *a* = Axis number    *p* = 0, .1, .2, .3, .4, .5

**compatibility:** MC200, MC210, MC260

**see also:**

Reports the position where the auxiliary encoder's index pulse was observed. This position is relative to the encoder's position when the controller was reset or an Auxiliary encoder define Home command was issued to the axis.

### Display the recorded Optimal position

**MCCL command:** aDO*p*    *a* = Axis number    *p* = integer > 0, <= 512

**compatibility:** MC200, MC210

**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.



## Display the recorded DAC output

**MCCL command:** aDQp    a = Axis number    p = integer > 0, <= 512

**compatibility:** MC200, MC210

**see also:** PR

This command is used to report the captured DAC output of an axis. See the description of **Record and display Motion Data** in the **Application Solutions** chapter.

## Display Recorded position

**MCCL command:** aDRp    a = Axis number    p = integer > 0, <= 512

**compatibility:** MC200, MC210

**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.

## Tell Analog

**MCCL command:** xTAp    x = Channel number    p = 0, .1, .2, .3, .4, .5

**compatibility:** MC500, MC510

**see also:**

Reports the digitized analog input signals to the DCX. The four 8-bit analog input channels accessed on connectors J3 are numbered 1,2,3 and 4. For each of these channels, the TA command will display 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 for the first four channels must be supplied to the DCX on connector J3, 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.

## Tell Breakpoint

**MCCL command:** aTBp    a = Axis number    p = 0, .1, .2, .3, .4, .5

**compatibility:** MC200, MC210, MC260

**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.

## Tell digital Channel

**MCCL command:** TCx    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".

## Tell Derivative gain

**MCCL command:** aTDp    a = Axis number    p = 0, .1, .2, .3, .4, .5

**compatibility:** MC200, MC210

**see also:** SD

Reports the derivative gain setting for a servo.

## Tell Contouring count

**MCCL command:** aTXp a = Axis number p = integer > 0, <= 2,147,483,647

**compatibility:** MC200, MC210, MC260

**see also:** CP

Reports the current contour path motion that an axis is performing. The value that the DCX replies is only valid for the controlling axis in a group of axes performing contoured path motion. After the Contour Mode command is issued to an axis, the TX command will have a reply value of 0. For each Linear or User Defined Contour Path motion that the controller completes, the contouring count will be incremented by one. For Arc Contour Path motions, the count will be incremented by 2. By counting the number of Contour Path commands that have been issued to the controller (1 for linear, 2 for arc), and comparing it to the response from the TX command, the user can determine on what segment of a continuous path motion the motors are on. The contour count is stored as a 32 bit value (2,147,483,647). To reset the contour count value and avoid 'wrap around', the user should stop motion and issue the Contour Mode command.

## Tell Following error

**MCCL command:** aTFp a = Axis number p = 0, .1, .2, .3, .4, .5

**compatibility:** MC200, MC210

**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.

## Tell proportional Gain

**MCCL command:** aTGp a = Axis number p = 0, .1, .2, .3, .4, .5

**compatibility:** MC200, MC210

**see also:** SG

Reports the proportional gain setting for a servo.

## Tell Integral gain

**MCCL command:** aTIp a = Axis number p = 0, .1, .2, .3, .4, .5

**compatibility:** MC200, MC210

**see also:** set Integral gain

Reports the integral gain setting for a servo.

## Tell integral Limit setting

**MCCL command:** aTLp a = Axis number p = 0, .1, .2, .3, .4, .5

**compatibility:** MC200, MC210

**see also:** IL

Reports the integral limit setting for a servo.

## Tell velocity Konstant

**MCCL command:** aTKp a = Axis number p = 0, .1, .2, .3, .4, .5

**compatibility:** MC200, MC210

**see also:** VG

Reports the velocity constant for a servo. This is the value that was set with the Velocity Gain command. For a closed loop stepper axis this command reports the Velocity Gain that was set during initialization.

## Tell Macros

**MCCL command:** TM $n$   $n$  = integer  $\geq -1$ ,  $\leq 1099$

**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, both in RAM and FLASH memory. See the description of **Macro Command** in the **DCX Operation** chapter.

## Tell Optimal

**MCCL command:** aTO $p$   $a$  = Axis number  $p = 0, .1, .2, .3, .4, .5$

**compatibility:** MC200, MC210, MC260

**see also:**

Reports the desired position for servos and current position for steppers. For servos, the reported value will be different than the position reported by the TP command if a following error is present.

## Tell Position

**MCCL command:** aTP $p$   $a$  = Axis number  $p = 0, .1, .2, .3, .4, .5$

**compatibility:** MC200, MC210, MC260

**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.

## Tell torQue

**MCCL command:** aTQ $p$   $a$  = Axis number  $p = 0, .1, .2, .3, .4, .5$

**compatibility:** MC200, MC210

**see also:** QM, SQ

Reports the current output for a servo module. See the description of the **Torque Mode Output Control** in the **Application Solutions** chapter.

## Tell Register 'n'

**MCCL command:** TR $n$   $n$  = integer  $\geq 0$ ,  $\leq 256$

**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.

**report the Status of an axis****MCCL command :** aTSp    a = Axis number    p = 0, .1, .2, .3, .4, .5**compatibility:**        MC200, MC210, MC260**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 than 32 results in formatted status displays. Status flags that are not valid for steppers are indicated by an asterisk. The meaning of each bit is listed below:

<b>Bit number</b>	<b>Description</b>
0	Busy (motor data being updated)
1	Motor On
2	At Target
3	Trajectory Complete (Optimal = Target)
4	Direction (0 = positive, 1 = negative)
5	Motor Jogging is Enabled
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	Unused
12	Breakpoint Reached (IP, IR, WP, WR)
13	Exceeded Max. Following Error *
14	Amplifier Fault Enabled *
15	Amplifier 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 (MC200, MC210)/Stepper Home (MC260)
25	Coarse home (current state)
26	Amplifier Fault *
27	Auxiliary Encoder Index
28	Limit Positive Input Active (current state)
29	Limit Negative Input Active
30	User Input 1 *
31	User Input 2 *

\* not valid for stepper modules

```
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 Hexidecimal 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
Jogging Disabled
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
Auxiliary Encoder Index = 0
```

Limit Positive Input = 1  
Limit Negative Input = 0  
User Input 1 = 0  
User Input 2 = 0

example:            1TS33

DCX returns:  
MOTOR AUXILIARY STATUS:  
Hard Motion Limit Mode = Turn Motor Off  
Soft Motion Limit Mode = Turn Motor Off  
Servo Loop Rate is Medium  
Synchronization is Off  
Servo Phasing is Standard  
Backlash Compensation is Off

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  
Dead band: 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  
Velocity override: 1.000000  
Module ADC Input 1: 0.019530    Input2: 0.000000

## Tell Target

**MCCL command:**    aTTp    a = Axis number    p = 0, .1, .2, .3, .4, .5

**compatibility:** MC200, MC210, MC260

**see also:**

Reports target position. This is the absolute position to which the servo or stepper motor 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.

## Tell Velocity

**MCCL command:** aTVp    a = Axis number    p = 0, .1, .2, .3, .4, .5

**compatibility:** MC200, MC210, MC260

**see also:**

Reports the current velocity of a servo or stepper motor. The value is reported in units of encoder counts per servo loop update.

## Tell index position

**MCCL command:** aTZp    a = Axis number    p = 0, .1, .2, .3, .4, .5

**compatibility:** MC200, MC210

**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.

## Tell firmware Version

**MCCL command:** VE

**compatibility:** N/A

**see also:**

Reports the revision level of the firmware in the ROM of the DCX. This command also displays the amount of memory installed on the DCX motion controller motherboard.

example:    VE

DCX returns:

DCX-AT200 Motion Controller

Hardware: 64K Dual Port RAM, 64K Private RAM, 256K Flash Memory

System Firmware Ver. PM1 Rev. 3.6a

Copyright (c) 1994-1999 Precision MicroControl Corporation

All rights reserved.

## I/O Commands

### Channel oFf

**MCCL command:** CFx      x = Channel number

**compatibility:** MC400

**see also:**

Causes digital I/O 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 than 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).

### Channel High

**MCCL command:** CHx      x = Channel number

**compatibility:** MC400

**see also:**

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 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 "low true".

### Channel In

**MCCL command:** Clx      x = Channel number

**compatibility:** MC400

**see also:**

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.

### Channel Low

**MCCL command :** CLx      x = Channel number

**compatibility:** MC400

**see also:**

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".

### Channel oN

**MCCL command:** CNx      x = Channel number

**compatibility:** MC400

**see also:**



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 than 0.4 volts DC).

## Channel Out

**MCCL command:** CTx      x = Channel number

**compatibility:** MC400

**see also:**

Used to configure digital I/O channel x as an output. The DCX will turn the channel "off" before changing it to an output.

## Get Analog

**MCCL command:** GAx      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.

## Output Analog

**MCCL command:** OAn      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.

## Reset Counter

**MCCL command:** RC

**compatibility:** N/A

**see also:**

Initializes DCX hardware pulse counter to 0.

**comment:** *Not implemented at this time.*

## Tell Analog

**MCCL command:** xTAp      x = Channel number      p = 0, .1, .2, .3, .4, .5

**compatibility:** MC500, MC510

**see also:**

Reports the digitized analog input signals to the DCX. The four 8-bit analog input channels accessed on connectors J3 are numbered 1,2,3 and 4. For each of these channels, the TA command will display 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 for the first four channels must be supplied to the DCX on connector J3, 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.

## Tell Channel

**MCCL command:** TCx      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".

## Macro and Multi-Tasking Commands

### Break

**MCCL command:** BK none

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.

### Escape Task

**MCCL command:** ET $n$   $n = \text{integer} \geq 0$

**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 **DCX Operation** chapter.

### Generate Task

**MCCL command:** GT $n$   $n = \text{integer} \geq 0, \leq 1099$

**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 **DCX Operation** chapter.

### Macro Call

**MCCL command:** MC $n$   $n = \text{integer} \geq 0, \leq 1099$

**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 **Macro Command** in the **DCX Operation** chapter.

### Macro Define

**MCCL command:** MD $n$   $n = \text{integer} \geq 0, \leq 1099$

**see also:** ET, GT, MD

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 **Macro Command** in the **DCX Operation** chapter.

On the DCX, macros can be stored in one of two types of memory. Macro numbers 0 through 9, and 256 through 1099 are stored in 'Flash' memory which is preserved even if power to the board is turned off. Macro numbers 10 through 255 are stored in static RAM memory. These macros will be erased if power to the board is turned off (unless a user supplied battery is connected to the board). A macro in the Flash memory has the disadvantage that it can't be redefined unless all the macros are erased. A macro in RAM can be redefined without erasing the existing macros, 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, and  $n$  is between 10 and 255 inclusive, the previously defined macro will be replaced by the new macro definition. If  $n$  is outside this range, the previously defined macro will be "undefined", but not replaced. Note that macro 0 will be executed when the board is powered up or reset.

## Macro Jump

**MCCL command:** MJ $n$   $n = \text{integer} \geq 0, \leq 1099$

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 **Macro Command** in the **DCX Operation** chapter.

## No Operation

**MCCL command:** NO none

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 $n$   $n = \text{integer } 0, 1, \text{ or } 2$

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. The parameter to this command selects which group of macros will be erased. A parameter value of 1, causes the macros in RAM to be erased, a value of 2 causes the macros in Flash memory to be erased, and a parameter value of 0 causes all macros to be erased. 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 **Macro Command** in the **DCX Operation** chapter.

## Tell Macros

**MCCL command:** TM $n$   $n = \text{integer} \geq -1, \leq 1099$

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 **Macro Command** in the **DCX Operation** chapter.

## Register Commands

### Accumulator Add

**MCCL command:** AAn  $n = \text{integer or real}$

Performs  $\text{ACC} = \text{ACC} + n$ , 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.

### Accumulator Complement, bit wise

**MCCL command:** AC none

Performs  $\text{ACC} = !\text{ACC}$ , the bit wise logical complement of the Accumulator (User Register 0). The result is stored in the Accumulator as a 32 bit integer.

### Accumulator Divide

**MCCL command:** ADn  $n = \text{integer or real}$

Performs  $\text{ACC} = \text{ACC}/n$ , 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.

### Accumulator logical Exclusive or with 'n', bit wise

**MCCL command:** AEn  $n = \text{integer or real}$

Performs  $\text{ACC} = \text{ACC} \wedge n$ , 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.

### Accumulator load

**MCCL command:** ALn  $n = \text{integer or real}$

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

### Accumulator Multiply

**MCCL command:** AMn  $n = \text{integer or real}$

Performs  $\text{ACC} = \text{ACC} * n$ , 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.

## Accumulator logical ‘aNd’ the ‘n’ , bit wise

**MCCL command:**     $ANn$      $n = \text{integer or real}$

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.

## Accumulator logical ‘Or’ with ‘n’ , bit wise

**MCCL command :**     $AO n$      $n = \text{integer or real}$

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.

## copy Accumulator to Register

**MCCL command:**     $ARn$      $n = \text{integer or real}$

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.

## Accumulator Subtract

**MCCL command:**     $ASn$      $n = \text{integer or real}$

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.

## copy Register to Accumulator

**MCCL command:**     $RAn$      $n = \text{integer or real}$

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.

## Accumulator Evaluate

**MCCL command:**     $AVn$      $n = \text{integer } \geq 0, \leq 25$

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 ( $1 \leq n \leq 25$ ) 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

## Get Analog

**MCCL command:** GAx 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.

## Get the module iD

**MCCL command:** GDx x = integer > 0, <= 6

Loads the accumulator with the type of motor module associated with an axis number

Module Type	ID code
MC200	0
MC210	16
MC260	1

## Get defaUlt axis

**MCCL command:** GUx x = integer > 0, <= 6

This command is used to place the current default axis number in the accumulator.

## Get the position of the auxiliary encoder

**MCCL command:** aGX a = Axis number

**compatibility:** MC200, MC210, MC260

This command reads the auxiliary encoder associated with axis a and places the value into the Accumulator (User Register 0).

## Output Analog

**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.

### Read the Byte at absolute memory location ' $n$ ' into the accumulator

**MCCL command:**  $aRBn$   $a$  = 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.

### Read the Double (64 bit real) value at absolute memory location ' $n$ ' into the accumulator

**MCCL command:**  $aRDn$   $a$  = 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.

### Read the Long (32 bit integer) value at absolute memory location ' $n$ ' into the accumulator

**MCCL command:**  $aRLn$   $a$  = 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.

### Read the float (32 bit real) value at absolute memory location ' $n$ ' into the accumulator

**MCCL command:**  $aRVn$   $a$  = 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.

### Read the Word (16 bit integer) value at absolute memory location ' $n$ ' into the accumulator

**MCCL command:**  $aRWn$   $a$  = 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.

### Shift Left accumulator by ' $n$ ' bits

**MCCL command:**  $SLn$   $n = \text{integer} > 0, \leq 31$

Performs  $ACC = ACC \ll n$ , 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.

### Shift Right accumulator by ' $n$ ' bits

**MCCL command:**  $SRn$   $n = \text{integer} > 0, \leq 31$

Performs  $ACC = ACC \gg n$ , 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.

### Tell Register ' $n$ '

**MCCL command:**  $TRn$   $n = \text{integer} \geq 0, \leq 256$

**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.

### Write the low Byte in the accumulator to absolute memory location ' $n$ '

**MCCL command:**  $WBn$   $n = \text{integer}$

This command will copy the low byte of the accumulator (User Register 0) to the byte located at absolute memory address  $n$ .

### Write the Double (64 bit real) value in the accumulator to absolute memory location ' $n$ '

**MCCL command:**  $WDn$   $n = \text{real}$

This command will copy a Double (64 bit real) from the accumulator (User Register 0 and 1) to absolute memory address  $n$ .

### Write the float (32 bit real) value in the accumulator to absolute memory location ' $n$ '

**MCCL command:**  $WVn$   $n = \text{real}$

This command will copy a float (32 bit real) from the accumulator (User Register 0) to absolute memory address  $n$ .

**Write the Long (32 bit integer) value in the accumulator to absolute memory location 'n'**

**MCCL command:**  $WLn$   $n = \text{integer}$

This command will copy a Long (32 bit integer) from the accumulator (User Register 0) to absolute memory address  $n$ .

**Write the low Word (16 bit integer) value in the accumulator to absolute memory location 'n'**

**MCCL command:**  $WWn$   $n = \text{integer}$

This command will copy the low Word (16 bit integer) of the accumulator (User Register 0) to absolute memory address  $n$ .

## Sequence (If/Then) Commands

## Do if channel oFf

**MCCL command:** DFX      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.

[illegible]

## Do if channel 'x' is on

**MCCL command:** DNx      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.

[illegible]

**If the accumulator is Below 'n', execute the next command, else skip 2 commands**

**MCCL command:**  $IB_n$   $n = \text{integer or real}$

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
```

**If bit 'n' of the accumulator is Clear, execute the next command, else skip 2 commands**

**MCCL command:** IC<sub>n</sub>      *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
```

## If the accumulator Equals “n”, execute the next command, else skip 2 commands

**MCCL command:** IEn       $n$  = 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 channel oFf do next command, else skip 2 commands

**MCCL command:** IFx       $x$  = 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
```

## If the accumulator is Greater than ‘n’ execute the next command, else skip 2 commands

**MCCL command:** IGn       $n$  = 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
```

## If channel ' oN do next command, else skip 2 commands

**MCCL command:** INx       $x$  = 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
```

## Interrupt on absolute Position

**MCCL command:** IPn       $n$  = integer or real

**compatibility:** MC200, MC210, MC260

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.

## Interrupt (set breakpoint reached flag) upon reaching relative position

**MCCL command:** IR $n$   $n$  = integer or real

**compatibility:** MC200, MC210, MC260

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.

## If bit 'n' of the accumulator is Set execute the next command, else skip 2 commands

**MCCL command:** IS $n$   $n$  = integer  $\geq 0$ ,  $\leq 31$

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
```

## If the accumulator is Unequal 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
```

## JumP 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
```

## Jump to command Relative

**MCCL command:** JRn  $n = \text{integer}$

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
```

## RePeat

**MCCL command:** RPn  $n = \text{integer} \geq 0, \leq 2,147,483,647$

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
```

## WaiT

**MCCL command:** WAn  $n = \text{integer or real} \geq 0$

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
```

## Wait for Edge

**MCCL command:** aWEx  $x = 0 \text{ or } 1$

**compatibility:** MC200, MC210, MC260

**see also:** FE

Wait until the coarse home input of a servo or closed loop stepper axis  $a$  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.

## Wait for digital channel oFf

**MCCL command:** WFx  $x = \text{Channel number}$

**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.

## Wait for encoder index mark

**MCCL command:** aWIn  $a = \text{Axis number}$   $n = \text{integer or real} \geq 0$

**compatibility:** MC200, MC210, MC260

**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.

## Wait for digital channel oN

**MCCL command:** WN<sub>x</sub>      *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.

## Wait for absolute Position

**MCCL command:** aWP<sub>n</sub>      *n* = integer or real

**compatibility:** MC200, MC210, MC260

**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.

## Wait for Relative position

**MCCL command:** aWR<sub>n</sub>      *n* = integer or real

**compatibility:** MC200, MC210, MC260

**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.

## Wait for Stop

**MCCL command:** aWS<sub>n</sub>      *n* = integer or real

**compatibility:** MC200, MC210, MC260

**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.

## Wait for Target

**MCCL command:** `aWTn`  $n = \text{integer or real}$

**compatibility:** MC200, MC210, MC260

**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'. The condition for a stepper motor to have reached its' target is that the controller has output the last step of the motion. The Wait for Target command should not be used for axes in contour mode.

```
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.



## Miscellaneous Commands

### set the RS-232 baud rate

**MCCL command:** BR*n*    *n* = integer 1, 2, 4, 8, 16, 32, or 64

**compatibility:** MF300

Used to change the programmed baud rate for the RS-232 interface. The actual baud rate is determined by using the value *n* in a countdown circuit. The values for parameter 'n' for standard baud rates is shown below:

Baud Rate	Parameter <i>n</i> =
19,200	1
9,600	2
4,800	4
2,400	8
1,200	16
600	32
300	64

The values given are decimal numbers. If you are in hex mode, be sure to enter the hexadecimal equivalent, or momentarily change to decimal mode.

example: BR8

;sets the baud rate to 2,400 baud

**comment:** This command takes effect immediately, so use with caution. Any value entered will result in some baud rate but not necessarily a standard one.

### Decimal Mode

**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 Hexidecimal Mode (HM) commands cannot be in the same command string.

### Echo oFf

**MCCL command:** EN*n*    *n* = 0, 1, 2, or 3

**compatibility:** MF300

**see also:** EF

Causes the DCX not to echo characters received through an ASCII command port.

Only data specifically requested from the DCX will be transmitted. Normally used when operating with the host or terminal in half duplex mode. Parameter *n* selects the terminating character or characters that will be transmitted with command replies. The following table lists the available options.

Parameter <i>n</i>	Terminating Characters
0	No change from current setting
1	Carriage Return (ASCII 13) Only
2	Linefeed (ASCII 10) Only
3	Carriage Return and Linefeed (ASCII 13 and 10)

## Echo oN

**MCCL command:** EFn      $n = 0, 1, 2, \text{ or } 3$

**compatibility:** MF300

**see also:** EN

Causes all characters received through an ASCII command port to be echoed to that port as received. Normally used when operating with a or terminal in full duplex mode. Parameter  $n$  selects the terminating character or characters that will be transmitted with command replies. The following table lists the available options.

Parameter $n$	Terminating Characters
0	No change from current setting
1	Carriage Return (ASCII 13) Only
2	Linefeed (ASCII 10) Only
3	Carriage Return and Linefeed (ASCII 13 and 10)

## Free Memory

**MCCL command:** FM

**see also:** ME

Returns the memory space allocated by the ME command and returns it to the 'heap'.

## display the supported MCCL commands

**MCCL command:** HE

**explanation:** Reports the valid DCX command mnemonics for the installed software version.

## Handshaking oFf

**MCCL command:** HF

**compatibility:** MF300

**see also:** HN

Disables hardware handshake of serial communications through the RS-232 module.

## Hexadecimal Mode

**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 Hexidecimal 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).

## Handshaking oN

**MCCL command:** HN

**compatibility:** MF300

**see also:** HF

Enables hardware handshake of serial communications through the RS-232 module.

## allocate MEmory

**MCCL command:** MEnn = integer >0, <= 8192

**see also:** FM

Formats and allocates scratch pad memory. The first allocated memory address will be loaded into the accumulator.

## Prompt Character

**MCCL command:** PCn n = integer >0, <= 255

This command sets the character that will be sent out an ASCII command port when the DCX completes execution of a command issued to that port. The parameter to this command is the ASCII code for the character. Issuing the command with a parameter of 0 will inhibit any character from being sent. The default prompt character is '>' (ASCII 62 decimal).

## Reset

**MCCL command:** aRT a = Axis number

**compatibility:** MC200, MC210, MC260, 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.

## disable RS-232 XON/XOFF protocol

**MCCL command:** XF

**compatibility:** MF300

**see also:** XN

This command disables the XON/OFF handshaking protocol of the RS-232 serial port. This command has no effect when issued through the host PC or IEEE-488 command interfaces.

## enable RS-232 XON/XOFF protocol

**MCCL command:** XN

**compatibility:** MF300

**see also:** XF

This command enables the XON/XOFF handshaking protocol of the RS-232 serial port. This command has no effect when issued through the host PC or IEEE-488 command interfaces.

## Chapter Contents

---

- DCX System Troubleshooting
- Communications Troubleshooting
- Troubleshooting – Tuning a Servo Motor
- Troubleshooting - Servo Motion chart #1
- Troubleshooting - Servo Motion chart #2
- Troubleshooting - Servo Motion chart #3
- Troubleshooting – Stepper Motion chart #1
- Troubleshooting – Limits and Home

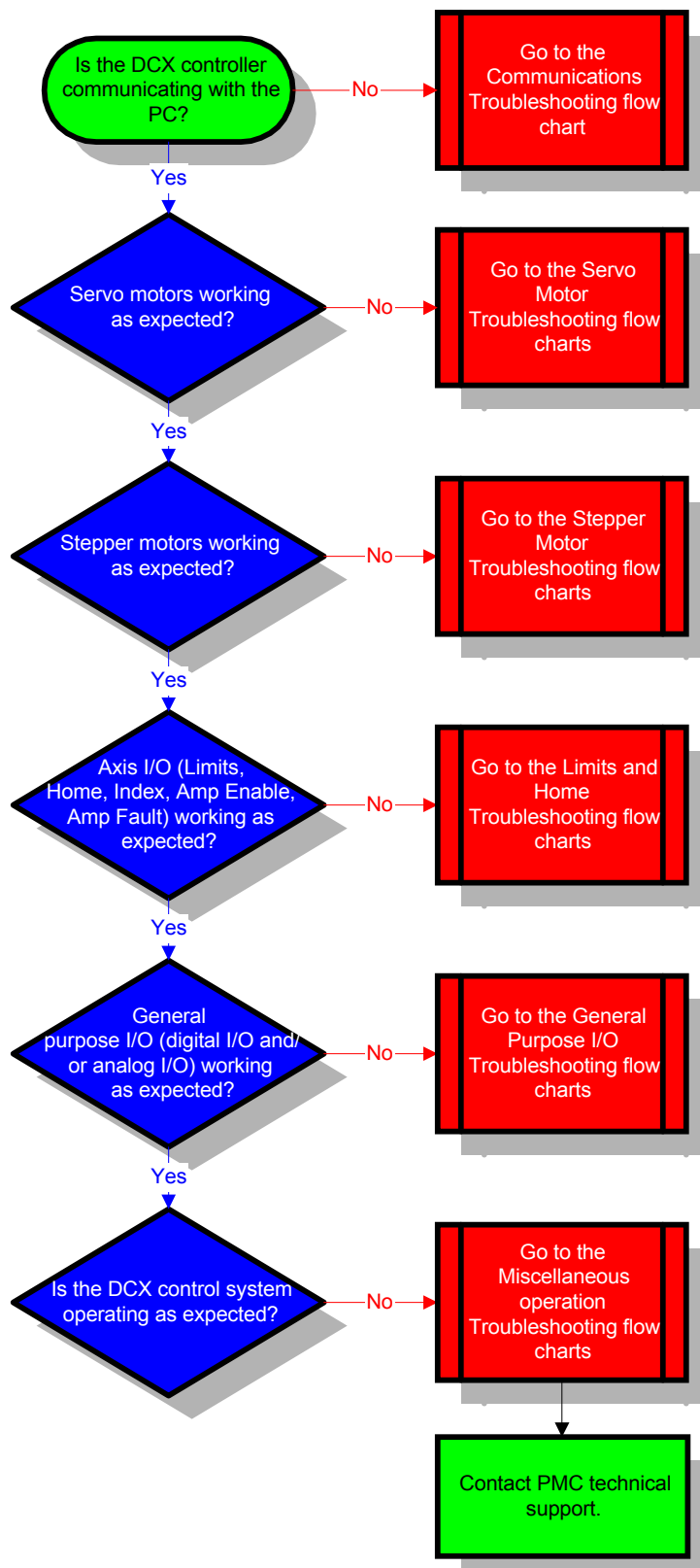
## Troubleshooting

---

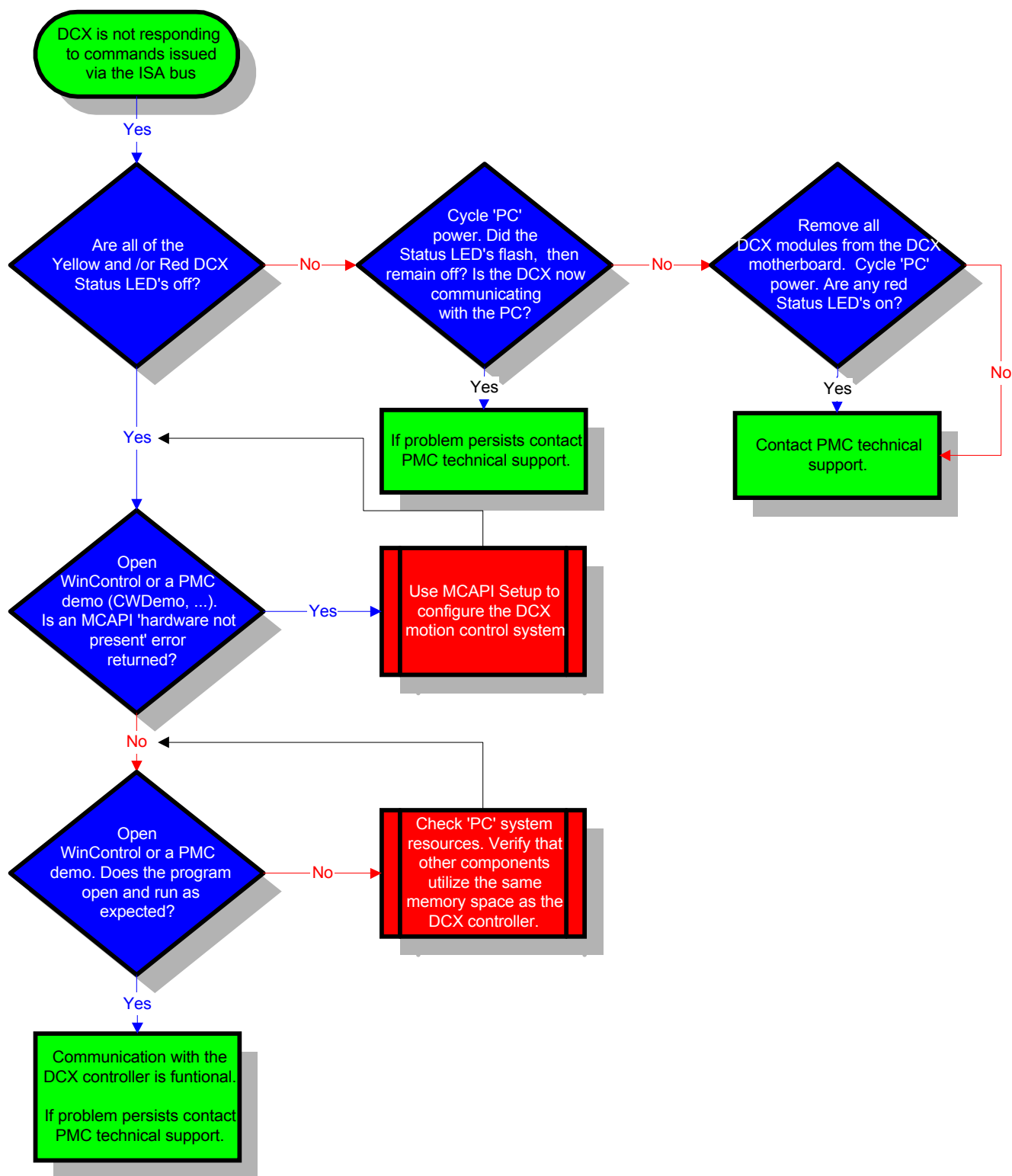
On the following pages you will find troubleshooting flow charts to assist in the with diagnosis of DCX system failures.

The steps described in these flow charts will direct the user to PMC programs (Motion Integrator, CWdemo, etc...) and utilities (Servo Tuning, WinControl) that are used to diagnose and resolve system performance.

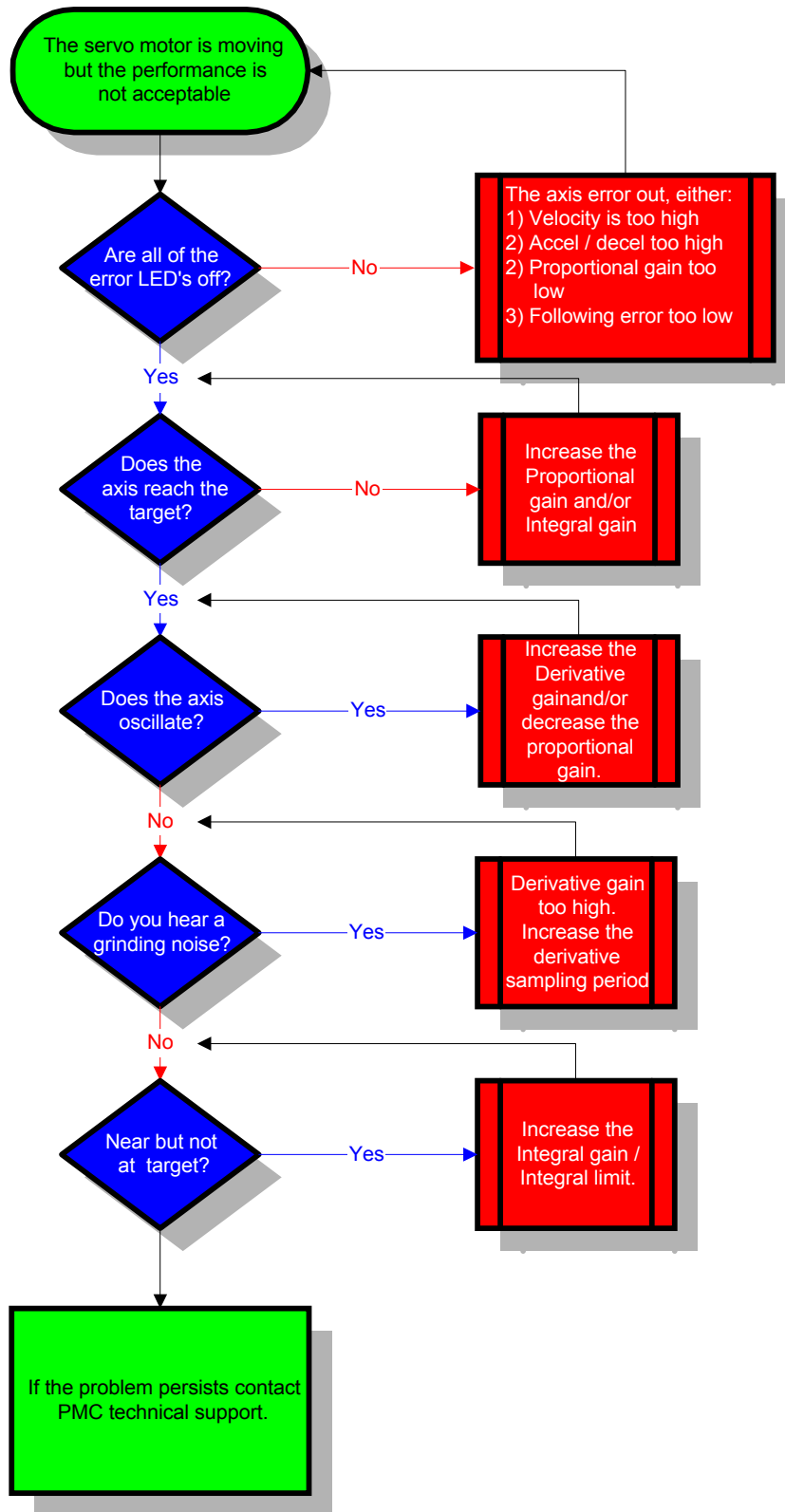
## DCX System Troubleshooting



# Communications Troubleshooting

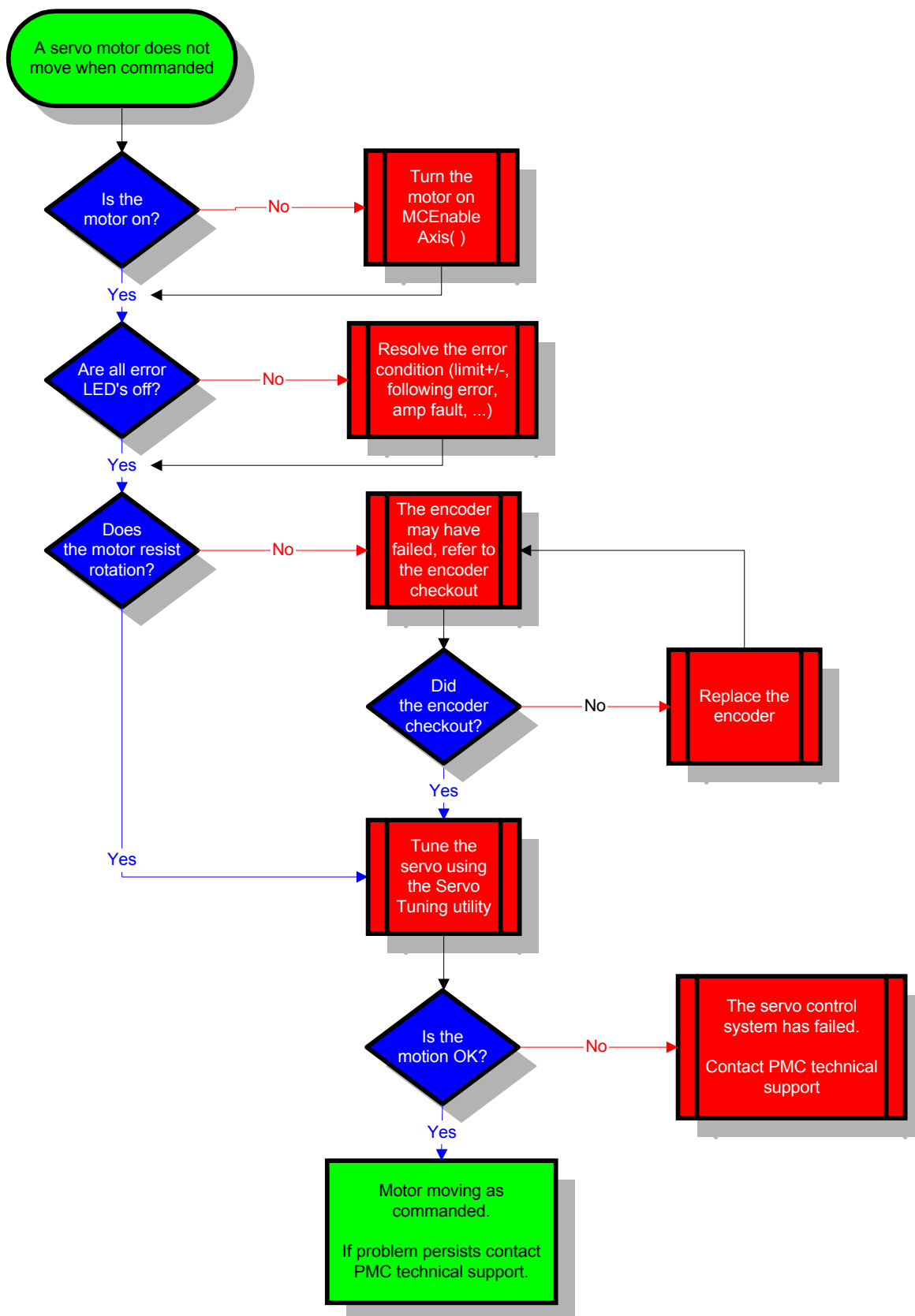


# Troubleshooting - Tuning a Servo Motor

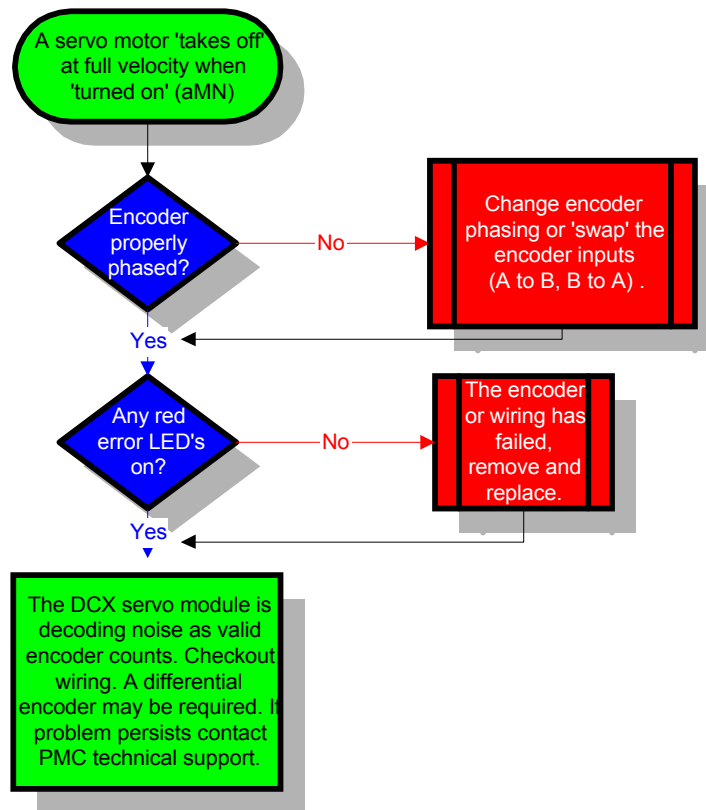




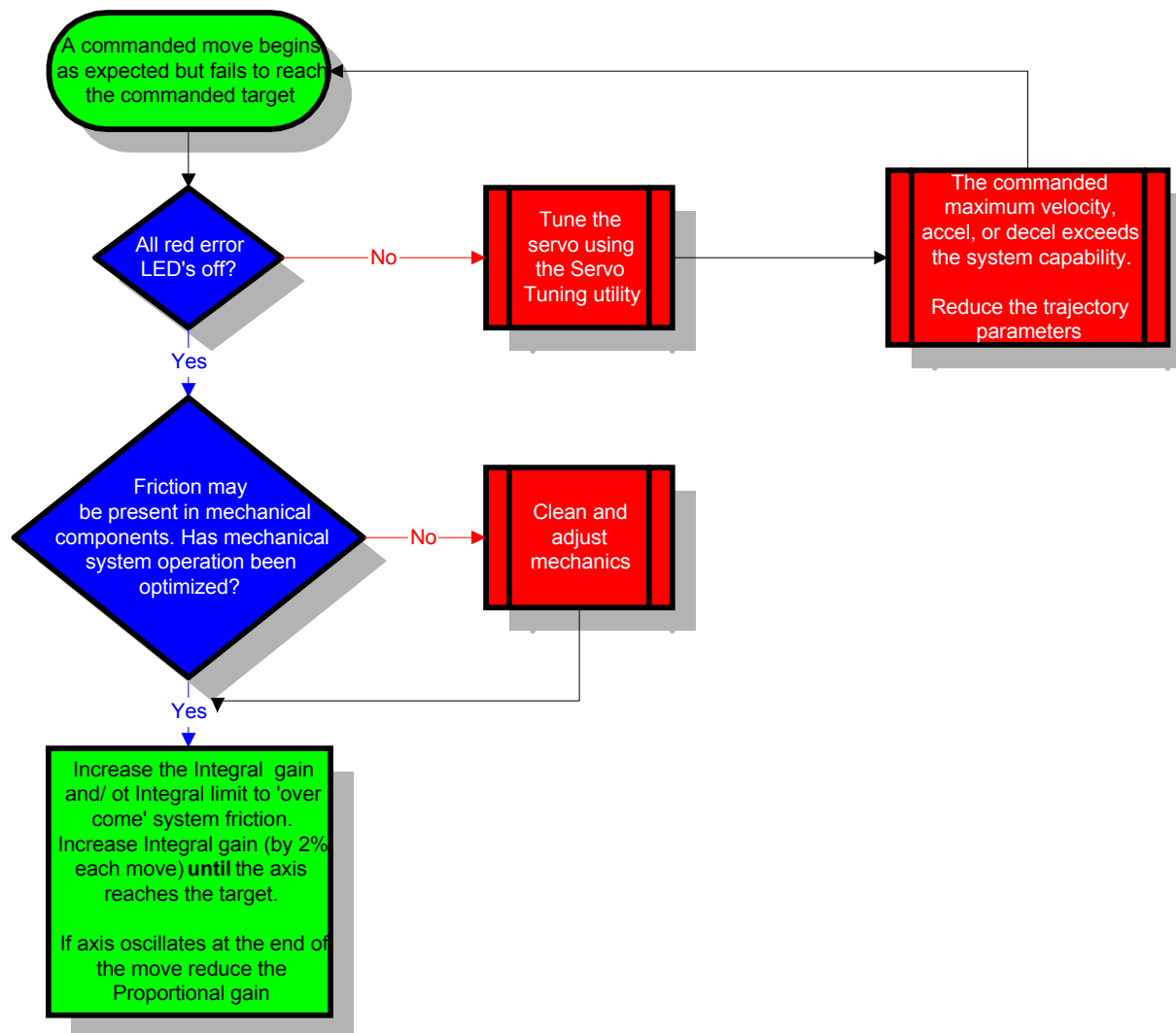
# Troubleshooting - Servo Motion chart #1



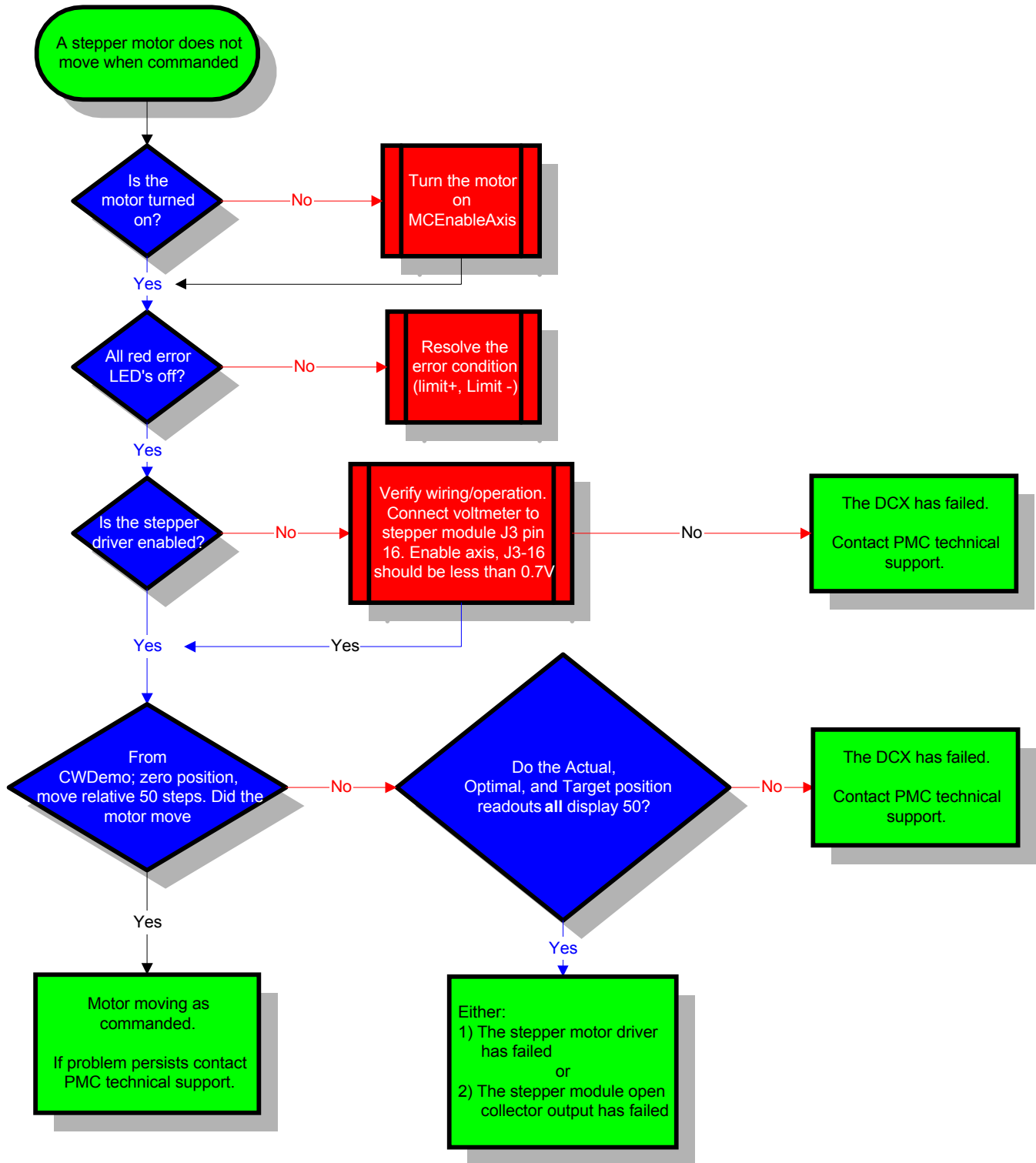
## Troubleshooting - Servo Motion chart #2



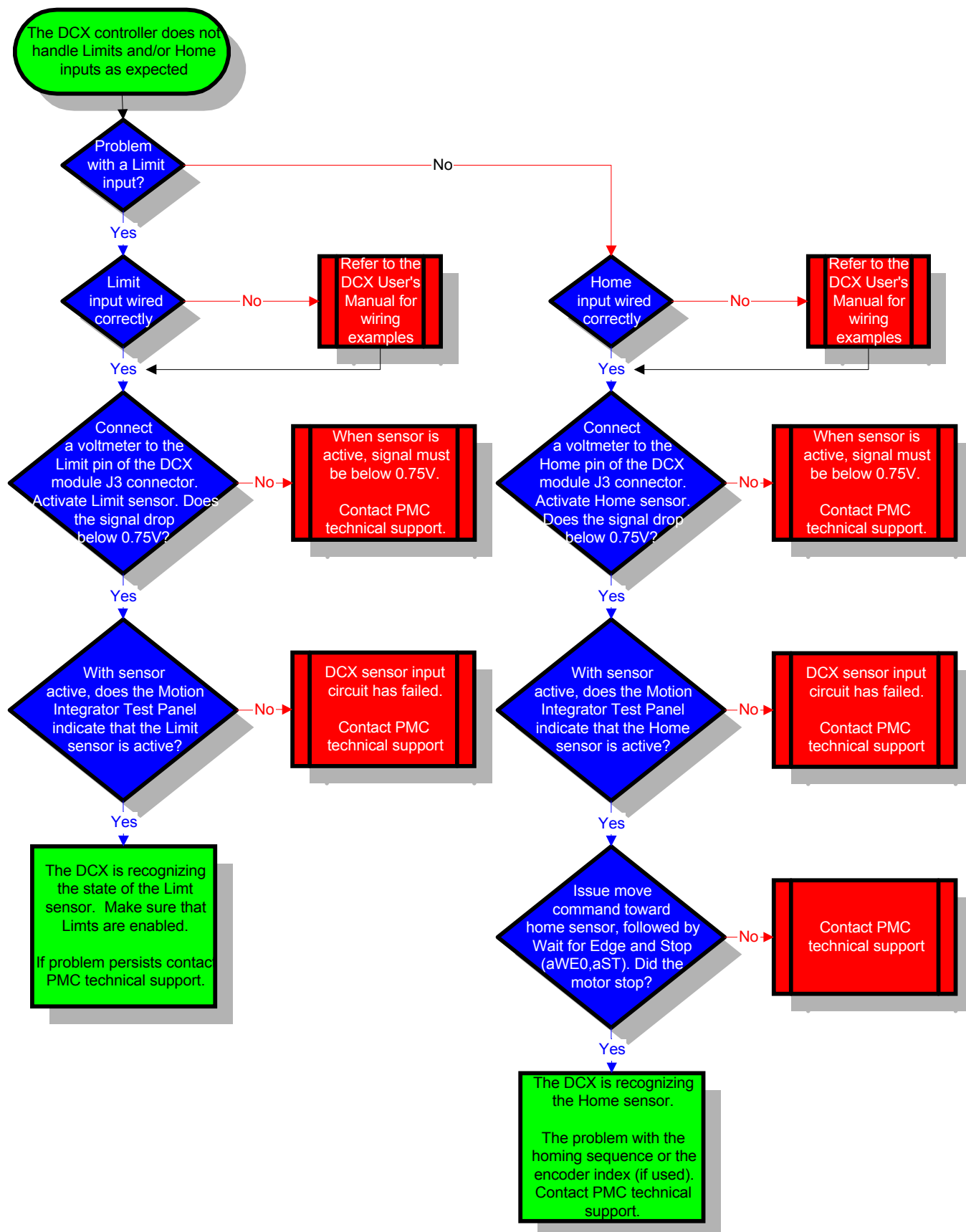
## Troubleshooting - Servo Motion chart #3



# Troubleshooting - Stepper Motion chart #1



# Troubleshooting - Limits and Home



## **Chapter Contents**

---

- MCAPI Error codes
- MCCL Error codes

## **Controller Error Codes**

---

Both the MCAPI and the Motion Control Command Language (MCCL) provide error code and interface status information to the user.

## MCAPI Error Codes

MCAPI defined error messages are listed numerically in the table below. 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.

<b>Error</b>	<b>Constant</b>	<b>Description</b>
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_WINDOWSEERROR	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_AXIS_TYPE	Axis type doesn't support this feature
14	MCERR_CONTROLLER	Invalid controller handle
15	MCERR_WINDOW	Invalid window handle
16	MCERR_AXIS_NUMBER	Axis number out of range
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)



## MCCL Error Codes

When executing MCCL (Motion Control Command Language) command sequences the command interpreter will report the following error code when appropriate:

<b>Description</b>	<b>Error code</b>
No error	0
Unrecognized command	1
Bad command format	2
I/O error	3
Command string too 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
Contour Path Command Not First	-8
Contour Path Command Parameter Invalid	-9
Contour Path Command Doesn't Specify an AXIS	-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.

The screenshot shows the WinControl32 application window. The command sequence entered is:

```
>RM
>1MN,2MN,3MN,MC10
?-6
[C3] MC10
>
>
>
>
```

Annotations on the screenshot:

- A red arrow points from the text "Error Code" to the "?-6" output line.
- A red arrow points from the text "Offending MCCL command" to the "[C3] MC10" input line.
- A red arrow points from the text "Position in command sequence" to the ">" prompt line immediately preceding the "[C3] MC10" input line.

## **Chapter Contents**

---

- Introduction to PDF
- Printing a complete PDF document
- Printing selected pages of a PDF document
- Paper
- Binding
- Pricing
- Obtaining a Word 2000 version of this user manual

## 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 printed 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.

## 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.

Micro-Stepping - Stepper drive systems have a fixed number of electromechanical detents or steps. Micro stepping is an electronic technique to break each detent or step into smaller parts. This results in higher positional resolution and smoother operation.

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

---

- Dual Ported Memory
- Binary Communication Interface
- ASCII Communication Interface
- Power Supply Requirements
- Pause and Resume Motion
- Physical Assignment of Axes
- Multiple User Communication Interfaces
- Stand Alone Applications
- On-Board File Operations
- HPGL Plotting

# Appendix

---

## Dual Ported Memory

When a DCX board is plugged in to a PC computer, depending on the setting of jumper JP8 and rotary switch SW1 on the DCX, a 4096 byte portion of the board's dual ported memory will be accessible somewhere in the PC's memory space. The location of this memory (defined by JP8), can be configured to be anywhere in the range 80000 hex to FFFFF hex.

### JP8 - IBM-PC Interface base memory address select

<b>Base address</b>	<b>JP8 5 to 6</b>	<b>JP8 3 to 4</b>	<b>JP8 1 to 2</b>
80000 hex	connected	connected	connected
90000 hex	connected	connected	open
A0000 hex	connected	open	connected
B0000 hex	connected	open	open
C0000 hex	open	connected	connected
<b>D0000 hex</b>	<b>open</b>	<b>connected</b>	<b>open</b>
E0000 hex	open	open	connected
F0000 hex	open	open	open

In most PC configurations, the 64K area from D0000 hex to DFFFF hex is available and has been chosen as the factory default memory range for the DCX. Most other ranges that can be set with jumper JP8 will result in hardware conflicts within the PC when the DCX is installed. Unless the user determines that another range is acceptable, jumper JP8 should be left at its default setting as shown in appendix B.

Assuming that jumper JP8 is at its factory default setting, the DCX will occupy a 4096 bytes somewhere between D0000 hex and DFFFF hex in the PC's memory space. If the

rotary switch SW1 is set to 0, the DCX will occupy D0000 hex through D0FFF hex. If it is set to 1, it will occupy D1000 hex through D1FFF hex, and so on. In the PC's memory map, the lowest numbered memory location that a DCX board occupies is referred to as the 'base' address. If the board's rotary switch is set to 0, its base address will be D0000h. The base address should be added to all addresses listed in this appendix in order to calculate the physical address in the PC's memory map.

**Motor Table Addresses:**

<b>Axis #</b>	<b>Motor Table Base Address</b>
1	0h
2	100h
3	200h
4	300h
5	400h
6	500h

Example: If the rotary switch is to 0, the position for axis 3 will be at D0214 hex.

**Motor Table Data:**

<b>Description</b>	<b>Data type</b>	<b>Offset</b>
Motor Status	unsigned integer	0 (0 hex)
Position Count	integer	4 (4 hex)
Optimal Count	integer	8 (8 hex)
Index Count	integer	12 (C hex)
Auxiliary Status	unsigned integer	16 (10 hex)

<b>Description</b>	<b>Data type</b>	<b>Offset</b>
Position	double	20 (14 hex)
Target	double	28 (1C hex)
Optimal Position	double	36 (24 hex)
Breakpoint Position	double	44 (2C hex)
Position Dead band	double	52 (34 hex)
Maximum Following Error	double	60 (3C hex)
Low Limit of Movement	double	68 (44 hex)
High Limit of Movement	double	76 (4C hex)
User Scale	double	84 (54 hex)
User Zero	double	92 (5C hex)
User Offset	double	100 (64 hex)
User Rate Conversion	double	108 (6C hex)
User Output Constant	double	116 (74 hex)
Programmed Velocity	double	124 (7C hex)
Programmed Acceleration	double	132 (84 hex)
Programmed Deceleration	double	140 (8C hex)
Minimum Velocity	double	148 (94 hex)
Current Velocity	double	156 (9C hex)

<b>Description</b>	<b>Data type</b>	<b>Offset</b>
Velocity Gain	float	164 (A4 hex)
Acceleration Gain	float	168 (A8 hex)
Deceleration Gain	float	172 (AC hex)
Velocity Override	float	176 (B0 hex)
Torque Limit	float	180 (B4 hex)
Proportional Gain	float	184 (B8 hex)
Derivative Gain	float	188 (BC hex)
Integral Gain	float	192 (C0 hex)
Integration Limit	float	196 (C8 hex)
Module Analog Input 1	float	200 (C4 hex)
Module Analog Input 2	float	204 (CC hex)
Jog Gain	float	208 (D0 hex)
Jog Offset	float	212 (D4 hex)
Jog Dead band		216 (D8 hex)
Jog Acceleration		220 (DC hex)
Jog Minimum Velocity		228 (E4 hex)

<b>Description</b>	<b>Data type</b>	<b>Offset</b>
Wait Stop Timer	short	326 (EC hex)
Wait Target Timer	short	238 (EE hex)
Sampling Frequency	short	240 (F0 hex)
Master Axis	short	242 (F2 hex)
Module Status	short	244 (F4 hex)
Axis Number	short	246 (F6 hex)
Module Position	short	248 (F8 hex)
Module Type	short	250 (FA hex)
Module Base Address	unsigned integer	252 (FC hex)

where:

double = 64 bit floating point format  
float = 32 bit floating point format  
integer = 32 bit integer format (2's complement)  
u.integer = 32 bit unsigned integer format  
short = 16 bit integer format

**Motor Status Bit Definitions:**

<b>Bit number</b>	<b>Description</b>
0	Busy (motor data being updated)
1	Motor On
2	At Target
3	Trajectory Complete (Optimal = Target)
4	Direction (0 = positive, 1 = negative)
5	Motor Jogging is Enabled
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	Unused
12	Breakpoint Reached (IP, IR, WP, WR)
13	Exceeded Max. Following Error *
14	Amplifier Fault Enabled *
15	Amplifier 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/Stepper Home
25	Coarse home (current state)
26	Amplifier Fault *
27	Auxiliary Encoder Index
28	Limit Positive Input Active (current state)
29	Limit Negative Input Active
30	User Input 1 *
31	User Input 2 *

\* not valid for stepper modules

**General use variables:**

<b>Description</b>	<b>Offset</b>
ASCII interface input mailbox	800h
ASCII interface output mailbox	804h
Command Interpreter status	808h
Position of Motor 1 Module	80Ah
Position of Motor 2 Module	80B4h
Position of Motor 3 Module	80Ch
Position of Motor 4 Module	80Dh
Position of Motor 5 Module	80Eh
Position of Motor 6 Module	80Fh
Type of Module in Position 1	812h
Type of Module in Position 2	813h
Type of Module in Position 3	814h
Type of Module in Position 4	815h
Type of Module in Position 5	816h
Type of Module in Position 6	817h
Digital I/O Channels 1-16	81Ah*
Analog Input Channel 1	81Ch*
Analog Input Channel 2	81Eh*
Analog Input Channel 3	820H*
Analog Input Channel 4	822H*

\* Value updated every millisecond

**Module type ID codes:**

<b>ID code</b>	<b>Module type</b>
0	MC200 Advanced Servo
1	MC260 Advanced Stepper
8	MC400 Digital I/O Module
9	MF310 GPIB Communications
10	MF300 RS-232 Communications
12	MC500 Analog I/O Module
15	No module present
16	MC210 Advanced servo, motor output

## Binary Communication Interface

In order to achieve the fastest PC to DCX command throughput, the DCX supports a "Binary" command interface. This interface allows the PC to write binary coded commands directly into a command buffer in the DCX's dual ported memory, thus bypassing the command interpreter. Using this method, any command replies the DCX generates will be returned in binary format to the PC through a reply buffer in the DCX's dual ported memory.

The MCAPI (Motion Control Application Programming Interface) supplied with the DCX on the utility disk, uses this command interface for high speed communication with the DCX. The high level programming languages supported by the MCAPI (C++, Visual Basic, Delphi, LabVIEW) all utilize this high speed communication interface.

The procedure for using the DCX Binary Command Interface is described below:

1. Beginning at the start of the command buffer in the DCX's dual port memory, write up to 255 bytes of commands. The number of commands that can be placed in the buffer depends on the format of each command.

A command that has no parameter will occupy 2 bytes in the buffer. Any commands that have a 32 bit integer or a 32 bit floating point parameter will occupy 6 bytes in the buffer. Any commands with a 64 bit double precision floating point parameter will occupy 10 bytes. A command with a string parameter can occupy whatever space is remaining in the command buffer. The host computer is free to use whichever number format is best suited to the command. The string format is reserved for specific commands that require a string parameter (see the command descriptions). Each command in the buffer must be in one of the following formats:

### **Command with no parameter**

**Byte 1:** Command code (listed in appendix A), if code is greater than 255, place 8 least significant bits in this byte and set extended command flag

**Byte 2:** Command flags & axis number\*

### **Command with 32 bit integer parameter**

**Byte 1:** Command code (listed in appendix A), if code is greater than 255, place 8 least significant bits in this byte and set extended command flag

**Byte 2:** Command flags & axis number\*

**Bytes 3-6:** Command parameter (binary integer, LSB first)

### **Command with 32 bit floating point parameter**

**Byte 1:** Command code (listed in appendix A), if code is greater than 255, place 8 least significant bits in this byte and set extended command flag

**Byte 2:** Command flags & axis number\*

**Bytes 3-6:** Command parameter (binary floating point, IEEE-754 format)

### **Command with 64 bit double precision floating point parameter**



**Byte 1:** Command code (listed in appendix A), if code is greater than 255, place 8 least significant bits in this byte and set extended command flag

**Byte 2:** Command flags & axis number\*

**Bytes 3-10:** Command parameter (binary double precision floating point, IEEE-754 format)

### **Command with string parameter**

**Byte 1:** Command code (listed in appendix A), if code is greater than 255, place 8 least significant bits in this byte and set extended command flag

**Byte 2:** Command flags & axis number\*

**Bytes 3-4:** String size in bytes including terminating null character

**Bytes 5-249:** String text followed by null (0 dec.) character

\* The command flags & axis number byte has the following format:

bits 0 - 3 specify the axis number, set these bits to 0 for all axes

bit 4 set if extended command (ie. code is greater than 255)

bits 5,6 & 7 specify the type of parameter the command has (see table below)

### **Parameter type bit coding:**

<b>bit 7</b>	<b>bit 6</b>	<b>bit 5</b>	<b>Parameter type</b>
0	0	0	32 bit integer parameter
0	0	1	64 bit floating point parameter
0	1	0	32 bit floating point parameter
0	1	1	No parameter (default value = 0)
1	0	0	32 bit integer specifying the user register that contains the parameter
1	0	1	string
1	1	0	Reserved
1	1	1	No parameter, use value in register 0

2. Write in the number of bytes written to the command buffer into the command count location.

3. Wait for the DCX to change the value in the command count location back to 0. This indicates that the commands in the buffer have been executed. At that time the DCX will cause the PC's interrupt line to go from low to high.

4. Any reply to the commands can be read from the reply buffer, with the value stored in the reply counter indicating the number of valid bytes. After reading the reply, the host computer should set the reply counter back to 0.

The format of the replies in the buffer is dependent on the command that generates the reply. The majority of the reporting commands use the command parameter to select the format of the reply. For these commands, a parameter value of 0 results in a 32 bit integer reply. A value of 1, results in a 64 bit floating point reply. And a value of 2 results in a 32 bit floating point reply. Commands such as Tell Register, Tell Channel, and Tell Analog which use the command parameter to select the item, have an implicit reply format. These formats are specified in the command description. In either case, a reply will have one of the following formats:

### **Reply with 32 bit integer value**

**Byte 1:** Command code (listed in appendix A), if code is greater than 255, 8 least significant bits will be in this byte and extended command flag will be set

**Byte 2:** Reply flags & axis number\*

**Bytes 3-6:** Reply value (binary integer, LSB first)

#### **Reply with 32 bit floating point value**

**Byte 1:** Command code (listed in appendix A), if code is greater than 255, 8 least significant bits will be in this byte and extended command flag will be set

**Byte 2:** Reply flags & axis number\*

**Bytes 3-6:** Reply value (binary floating point, IEEE-754 format)

#### **Reply with 64 bit double precision floating point value**

**Byte 1:** Command code (listed in appendix A), if code is greater than 255, 8 least significant bits will be in this byte and extended command flag will be set

**Byte 2:** Reply flags & axis number\*

**Bytes 3-10:** Reply value (binary double precision floating point, IEEE-754 format)

#### **Reply with string text**

**Byte 1:** Command code (listed in appendix A), if code is greater than 255, 8 least significant bits will be in this byte and extended command flag will be set

**Byte 2:** Reply flags & axis number\*

**Bytes 3-4:** String size in bytes including terminating null character

**Bytes 5-249:** String text followed by null (0 dec.) character

\* The reply flags & axis number byte will have the following format:  
 bits 0 - 3 specify the axis number, these bits will be 0 for no axis  
 bit 4 set if extended command (ie. code is greater than 255)  
 bit 5,6 & 7 indicate the type of reply value (see table below)

#### **Reply value type bit coding:**

<b>bit 7</b>	<b>bit 6</b>	<b>bit 5</b>	<b>Parameter type</b>
0	0	0	32 bit integer parameter
0	0	1	64 bit floating point parameter
0	1	0	32 bit floating point parameter
0	1	1	Reserved
1	0	0	Reserved
1	0	1	string
1	1	0	Reserved
1	1	1	Reserved

5. If it is necessary for the PC to terminate command execution before the DCX has set the command count back to 0, write a binary FFh into the command counter.

6. In order to clear the interrupt line, the PC can issue the next group of commands or write a binary FFh to the command counter location.

The address offsets of the command and reply buffers in the DCX's dual port memory are as follows:

<b>Description</b>	<b>Memory address offset</b>
Command Count	900h
Command Buffer	901 – 9FFh
Reply Count	A00h
Reply Buffer	A01 - AFFh

## ASCII Communication Interface

In order for the host computer to communicate to the DCX in ASCII characters, two mailbox locations have been defined. The data input mailbox is used to send data to the DCX. This mailbox is the byte located at 800 hex from the base address. For a DCX controller with the default configuration, this will be absolute address D0800 hex. The data output mailbox is used to receive data from the DCX. This mailbox is the byte located at 804 hex from the base address. This will be absolute address D0804 hex.

A simple protocol allows ASCII characters to be transferred to and from DCX boards. In order to send an ASCII character to a board, first read the contents of its data input mailbox, if it is 0, write the data into the mailbox. If the contents are non-zero, the previous data put in the mailbox has not yet been processed by the DCX board. The host should continue checking the mailbox until the contents become 0, and then write the data.

In order to receive ASCII characters from a DCX board, read the contents of its data output mailbox. If the value read is non zero, it is a valid character and the host should save the character and then write a 0 to the data output mailbox to indicate it has received the data. If the contents of the mailbox is 0, the DCX has not placed new data in the mailbox. In this case the host can continue checking the mailbox for new data.

In addition to the command interfaces, the host can read motor information directly from the DCX's memory. An appendix of this manual details the contents of the "DCX Dual Port RAM".

## Power Supply Requirements

<b>Part Number</b>	<b>+5 VDC</b>	<b>+12 VDC</b>	<b>-12 VDC</b>	<b>Unit</b>
DCX-AT200	0.9	-----	-----	A
DCX-MC200	.2	.01	.01	A
DCX-MC210	.2	*	*	A
DCX-MC260	.2	-----	-----	A
DCX-MC400	.25	-----	-----	A
DCX-MC500	.1	*	*	A
DCX-MF300	.01	*	*	A
DCX-MF310	.16	-----	-----	A

\* Current depends on output loading

## Default Settings

<b>Description</b>	<b>Setting</b>
Programmed Velocity	10,000
Programmed Acceleration	10,000
Programmed Deceleration	10,000
Minimum Velocity	1,000
Current Velocity	0
Velocity Gain	0
Acceleration Gain	0
Deceleration Gain	0
Velocity Override	1
Torque Limit	10
Proportional Gain	.2
Derivative Gain	.1
Integral Gain	.01
Integration Limit	50
Maximum Following Error	1024
Motion Limits	disabled
Low Limit of Movement	0
High Limit of Movement	0
Servo Loop Rate	MS
Stepper Pulse Range	HS
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
Jog Gain	1000
Jog Offset	2.5
Jog Dead band	.1
Jog Acceleration	1000
Jog Minimum Velocity	0
Sampling Frequency	0
Slave Ratio	1

## Pause and Resume Motion

The Save Configuration (*aSCn*) and Restore Configuration (*aRCn*) commands can be used with the Velocity Override command to pause and resume motion.

Each of these commands takes an axis specifier *a* and requires a file number as the command parameter *n*. These commands save and restore the entire motor table. This includes the public motor table in dual port memory and the private motor table in internal RAM.

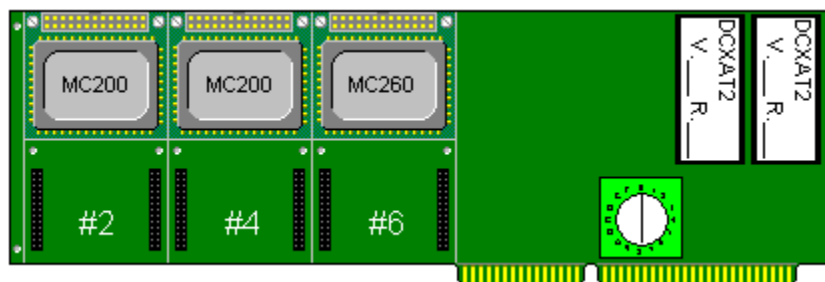
These commands allow the motors to be stopped (*aVO0*) during a contour move, their configurations saved, switched to any other mode (except contouring), moved about and then returned to their original positions, their configurations restored, and then commanded to continue the contour move (*aVO1.0*).



Note: Prior to resuming motion it is very important that the axes be returned to the **exact** position at which the motor table was saved. If this is not done, the axis will either jump to the position at which motion was paused or it may error out.

## Physical Assignment of Axes Numbers

The DCX defaults to assigning axis numbers logically, not based solely on a motor module's physical location. In the graphic below three modules are installed on a DCX-AT200. The MC200 in module location #1 would be defined as axis one. The MC200 in module location #3 would be defined as axis two. The MC260 in module location #5 would be defined as axis three.



The DCX does support the assigning of axis numbers based only on their physical location. The Use Physical command will redefine the logical addresses. To redefine axes 2 and 3 in the graphic above as axes 3 and 5:

---

```

3UP3                                ;Define the module in location 3 as
                                    ;axis 3
5UP5                                ;Define the module in location 5 as
                                    ;axis 5

```



Note – The reassignment of axes should be done before sending any other commands (setup, move, etc...).

## Multiple User Communications Interfaces

The DCX supports multiple user interfaces. The interfaces supported are:

- ISA Bus Binary
- ISA Bus ASCII
- Serial ASCII (requires installation of one MC300 RS-232 module)
- GPIB ASCII (Requires installation of one MF310 IEEE-488 module)

Each of the four available command interfaces are handled by a separate CPU task controller. Any or all user interface tasks can run simultaneously and independently.

Multi user interfaces offer the machine programmer great flexibility in supporting messaging. A typical example of this capability would be a Vertical Form Fill and Seal (VFFS) packaging application. After an empty bag is filled with coffee, a 'hot seal bar' is used to close the bag. Most machine operations would typically be programmed using PMC's Motion Control API which communicates via the binary interface. But monitoring the temperature of the hot seal bar could be implemented via a DCX macro running as a background task. This would allow the operator interface to display a preprogrammed ASCII text message issued from the DCX if the seal bar is not within the proper temperature range. For additional information on messaging please refer to the section **titled Outputting Formatted Messages**.

The Command interpreter status word in dual port memory (808h offset) has the following format:

<b>Bit</b>	<b>Flag</b>
0	Host Binary Interface - Busy
1	Unused
2	Unused
3	Unused
4	Host Ascii Interface - Busy
5	Unused
6	Unused
7	Host Ascii Interface - Loading File
8	Serial Interface - Busy
9	Unused
10	Unused
11	Serial Interface - Loading File
12	GPIB Interface - Busy
13	Unused



---

14	Unused
15	GPIB Interface - Loading File

## Stand Alone Applications

When used in the stand-alone configuration, power must be provided to the DCX through the edge connector. In this configuration, one of the available communication interface modules (RS-232 or IEE-488) can be used to send commands to the controller. In this case, a mating edge connector with specific pins grounded or pulled high (connected to the 5 volt supply through a resistor) should be used to properly disable the host interface. Please refer to the Stand Alone Edge Connector description in Appendix A.

## On-Board File Operations

The DCX has the ability to store text files in its on-board memory. The primary purpose for this capability is for support of the Plotting function described in a later section. This section describes the commands used to load and manage files in the controller's file system.

The standard memory on the DCX provides 8 Kbytes (8,192 characters) of memory for file storage. With the expanded memory option, an additional 191 Kbytes (195,584 characters) of file storage is available. The file system organizes the memory using a directory that can maintain up to 127 separate files. Each file is identified by an integer number from 1 to 127. A file can be any size, up to the available memory in the file system. A file can contain any number of ASCII characters, but the total space required by all files cannot exceed the file system memory. Note that file storage space is allocated in 1024 byte blocks, so a file that is 100 bytes will occupy 1024 bytes of file system memory.

Prior to loading any files onto the DCX controller, the file memory must be initialized, similar to formatting a floppy disk of a personal computer. This is accomplished by issuing the Format command to the DCX.

Example:FO

Initializing the file system memory will delete all existing files from memory and make all space available for new files.

Files can be loaded into the controller using any of the ASCII interfaces. Files can't be loaded using the binary interface. To load a file, issue the L`OAD` command with a parameter from 1 to 127 specifying the file to be loaded. After issuing this command the board will accept all characters received at the interface and place them in the file. To terminate loading of the file, send the End-Of-File ASCII character (1A hex) to the interface. This code can be generated by pressing the control-Z key combination on the PC keyboard. When a file is loaded in this manner, all previous contents of the specified file will be lost.

A program on the Utility Disk, DCX2LOAD can be used to copy a file on the host computer's disk drive into the DCX's file storage. This DOS program takes the name of the file as a command line parameter. Include a command line parameter with the form /C"LO`n`", where `n` is a number between 1 and 127 specifying the file number on the DCX that the file will be copied to. The program will terminate immediately after copying the file.

Example:DCX2LOAD /C"LO10" FILE.EXT

This DOS command line will copy a file named 'FILE.EXT' to the DCX and store it as file number 10.

To display what has been loaded into a file, issue the T`Y`pe command to the controller from one of the ASCII interfaces. The controller will respond by displaying the entire contents of the file. To pause the file display, press the Ctrl-S key combination. To resume the display, press the Ctrl-Q key combination.

The Directory Listing command is used to display a list of what files are loaded and their respective sizes.

The Remove File command issued with a parameter value from 1 to 127 will delete the respective file from the controller's memory.

The following DCX commands are available for creating and maintaining files in the controller's memory:

DL                Directory Listing

syntax:          DL

code:            271d, 10Fh

parameter:      none

compatibility:   N/A

see also:        TY

explanation: This command causes the DCX to display information about the files that are currently loaded in its memory and the amount of file storage space remaining for new files.

example:    DL

FILE 1: SIZE = 8377

FILE 10: SIZE = 6129

FILE 100: SIZE = 26574

FREE SPACE = 162696 BYTES

EC                Execute Command file

syntax:          EC

code:            279d, 117h

parameter:      1 <= n <= 127

compatibility:   N/A

see also:

explanation: Implemented Execute Command file (EC, code=279) command. This command assumes the on-board file specified by the EC command parameter contains DCX commands. If single stepping is enabled while executing the command file, the file and line numbers will be displayed with the command index.

FO                FOrmat file system

syntax:          FO

code:            270d, 10Eh

parameter:      none

compatibility:   N/A

see also:        RF

explanation: This command initializes the DCX's file storage memory and makes the full amount of memory allocated for file storage available. Any files that are stored in the file system will be erased when this command is executed.

LO                LOn file

syntax:          LOn

code:            274d, 112h

parameter:      integer (1 <= n <= 127)

compatibility:   MC200, MC260

see also:        TY

explanation: When this command is issued over one of the DCX's ASCII interfaces, the controller will begin accepting characters from that interface and storing them sequentially in the file specified by the command parameter. Loading of the file will be terminated when the End-Of-File ASCII character (1A hex) is received.

RF                Remove File  
syntax:          RFn  
code:            273d, 111h  
parameter:      integer (1 <= n <= 127)  
compatibility:   N/A  
see also:        DL

explanation: This command will delete the specified file from the DCX's storage memory. Functionally this command will make the memory occupied by the specified file available for storage of new files, and set the directory entry for this file to 'empty'.

TY                TYpe file  
syntax:          aCMn  
code:            272d, 110h  
parameter:      integer (0 <= n <= 127)  
compatibility:   N/A  
see also:        DL

explanation: This command will cause the contents of the specified file to be sent to the ASCII interface that the command was issued to.

example:    TY10

```
IN;SP1;  
PA1000,0;  
PA1000,1000;  
PA0,1000;  
PA0,0;
```

## HPGL Plotting

The DCX has the ability to execute Hewlett Packard Graphic Language (HPGL) commands. This language is a popular standard used for controlling X-Y tables in plotting and engraving applications. The DCX can accept HPGL commands sent to it over an ASCII interface port, or it can read them from an on-board file (see **On-Board File Operations**). Prior to executing HPGL commands from either source, the plotting environment must be setup by issuing DCX plotter configuration commands.

After configuring the plotting environment, the Plotting Enable command will cause the DCX to accept HPGL commands from the ASCII interface port that the PE command was issued to. Once plotting is enabled, the DCX will accept only HPGL formatted commands until plotting is disabled by sending it an End-Of-File ASCII character (1A hex). Plotting can be re-enabled without issuing the configuration commands if no changes are required.

Alternatively, HPGL commands can be stored in an on-board file and the Plot File command issued to the DCX using the file number as the command parameter. In this mode, the DCX will execute the HPGL commands in the file without host computer assistance. Similar to the Plot Enable command, the Plot File command can be issued multiple times without issuing the configuration commands if no changes are required.

A program on the Utility Disk, DCX2LOAD can be used to plot a file on the host computer's disk drive. This DOS program takes the name of the file as a command line parameter. Include a command line parameter with the form /C"PE". The program will terminate immediately after plotting the file.

Example:DCX2LOAD /C"PE" FILE.EXT

This DOS command line will plot a file named 'FILE.EXT' using the DCX.

### DCX Plotter configuration commands:

PA                Plotter Acceleration

syntax:          PAn

code:            289d, 121h

parameter:      integer or real (0 <= n)

compatibility:   MC200, MC260

see also:        PV

explanation: This command sets the acceleration and deceleration for contoured motion of the X and Y plotting axes. The default value for the plotting acceleration is 1.0.

PD                Pen Down

syntax:          PDn

code:            293d, 125h

parameter:      integer or real (0 <= n <= 1099)

compatibility:   N/A

see also:        PU,SP

explanation: Specifies a macro containing DCX commands that will be called whenever the HPGL Pen Down (PD) command is executed.

PE                    Plotting Enable  
syntax:            PE  
code:              281d, 119h  
parameter:        none  
compatibility:    MC200, MC260  
see also:          PF

explanation: This command places the DCX in plotting mode. Once this command is executed, the board will only accept HPGL commands from the command port that the PE command was issued to. To terminate plotting, the End-Of-File ASCII character (1A hex) must be sent to the same port.

PF                    Plot File  
syntax:            PFn  
code:              280d, 118h  
parameter:        integer (1 <= n <= 127)  
compatibility:    MC200, MC260  
see also:          PE

explanation: This command will cause the specified file to be plotted. See the **On-Board File Operation** section for an explanation of how to load a file prior to issuing this command. This command can be executed as either a foreground or background task.

PI                    Plotter Initialize macro  
syntax:            PIn  
code:              291d, 123h  
parameter:        integer (0 <= n)  
compatibility:    N/A  
see also:          PU,PD,SP

explanation: Specifies a macro containing DCX commands that will be called whenever the HPGL Initialize (IN) command is executed.

PQ                    Plotter Quick velocity  
syntax:            PQn  
code:              290d, 122h  
parameter:        integer or real (0 <= n)  
compatibility:    MC200, MC260  
see also:          PV

explanation: This command sets the velocity for Pen Up motion of the X and Y plotting axes. The parameter to this command will be in the plotting units that were set by the Plotter X and Y Scaling commands.

PU                    Pen Up macro  
syntax:            Un  
code:              292d, 124h  
parameter:        integer (0 <= n <= 1099)  
compatibility:    N/A  
see also:          PD,SP

explanation: Specifies a macro containing DCX commands that will be called whenever the HPGL Pen Up (PU) command is executed.

**PV**                      Plotter Velocity  
syntax:                PVn  
code:                  288d, 120h  
parameter:           integer or real ( $0 \leq n$ )  
compatibility: MC200, MC260  
see also:             PA,PQ  
explanation: This command sets the velocity for the Pen Down contoured motion of the X and Y plotting axes. The parameter to this command will be in the plotting units that were set by the Plotter X and Y Scaling commands. Alternatively, the parameter to this command will be used as a scaling factor if a HPGL Velocity Select command is executed during plotting. The default value for the plotting velocity is 1.0.

**PX**                      Plotter X axis  
syntax:                PXn  
code:                  282d, 11Ah  
parameter:           integer ( $0 \leq n \leq 6$ )  
compatibility: MC200, MC260  
see also:             PY  
explanation: Specifies the DCX controller axis that will be used for the X axis motion in executing the HPGL commands.

**PY**                      Plotter Y axis  
syntax:                PYn  
code:                  283d, 11Bh  
parameter:           integer ( $0 \leq n \leq 6$ )  
compatibility: MC200, MC260  
see also:             PX  
explanation: Specifies the DCX controller axis that will be used for the Y axis motion in executing the HPGL commands.

**SP**                      Select Pen macro  
syntax:                SPn  
code:                  294d, 126h  
parameter:           integer ( $0 \leq n \leq 1099$ )  
compatibility: N/A  
see also:             PU,PD  
explanation: Specifies a macro containing DCX commands that will be called whenever the HPGL Select Pen (SP) command is executed. Prior to calling the macro, the pen number that was specified with the HPGL Select Pen command will be placed in User Register 0. The macro can use the pen number stored in User Register 0 to determine which pen or tool to select.

**XO**                      plotter X Offset  
syntax:                XOn  
code:                  286d, 11Eh  
parameter:           integer or real  
compatibility: MC200, MC260  
see also:             XS,YO

explanation: Specifies an offset in the origin for the plotting X axis. The parameter to this command should be in the same plotter units that are established with the Plotter X Scale (XS) command. The value set with this command will be used as the axes' User Offset when the Plotting Enable (PE) command is issued. This offset is independent of the offset that can be set with the HPGL Input P1 and P2 (IP), and Scale (SC) Commands. The default offset is 0.

XS                plotter X Scale  
syntax:        XS<sub>n</sub>  
code:          284d, 11Ch  
parameter:    integer or real  
compatibility: MC200, MC260  
see also:      XO,YS

explanation: Specifies the scaling factor for the plotting X axis. The parameter to this command should be the number of encoder counts or steps per plotter unit. The value set with this command will be used as the axes' User Scale when the Plotting Enable (PE) command is issued. This scaling is independent of the scaling that can be set with the HPGL Input P1 and P2 (IP), and Scale (SC) Commands. The default scale factor is 1.0, or 1 plotter unit per encoder count or step.

YO                plotter Y Offset  
syntax:        YOn  
code:          287d, 11Fh  
parameter:    integer or real  
compatibility: MC200, MC260  
see also:      XO,YS

explanation: Specifies an offset in the origin for the plotting Y axis. The parameter to this command should be in the same plotter units that are established with the Plotter Y Scale (YS) command. The value set with this command will be used as the axes' User Offset when the Plotting Enable (PE) command is issued. This offset is independent of the offset that can be set with the HPGL Input P1 and P2 (IP), and Scale (SC) Commands. The default offset is 0.

YS                plotter Y Scale  
syntax:        PAn  
code:          285d, 11Dh  
parameter:    integer or real  
compatibility: MC200, MC260  
see also:      XS,YO

explanation: Specifies the scaling factor for the plotting Y axis. The parameter to this command should be the number of encoder counts or steps per plotter unit. The value set with this command will be used as the axes' User Scale when the Plotting Enable (PE) command is issued. This scaling is independent of the scaling that can be set with the HPGL Input P1 and P2 (IP), and Scale (SC) Commands. The default scale factor is 1.0, or 1 plotter unit per encoder count or step.



---

## Supported HPGL PLOTTING commands

When plotting is configured (using the DCX plotting configuration commands) and then enabled on the DCX, it will accept and execute Hewlett Packard Graphic Language (HPGL) commands. This appendix describes the HPGL commands that have been implemented on the DCX.

In the syntax descriptions, upper case letters are required, lower case letters represent numerical values, and parenthesis delimit optional values.

### Arc Absolute

syntax: AA x\_abs,y\_abs,angle (,chord)

### Arc Relative

syntax: AA x\_rel,y\_rel,angle (,chord)

### Circle

syntax: CI radius

### INInitialize

syntax: IN

### IP Input P1 and P2

syntax: IP p1x,p1y(,p2x,p2y)

### PA Plot Absolute

syntax: PA x,y(...)

### PD Pen Down

syntax: PD (x,y(...))

### PR Plot Relative

syntax: PR x,y(...)

### PU Pen Up

syntax: PD (x,y(...))

### SC Scale

syntax: SC xmin,xmax,ymin,ymax

### SP Select Pen

syntax: SP n

### VS Velocity Select

syntax: VS v



# Index

## A

Abort motion	178
Acceleration	
disable	50
setting	48, 85, 87, 93, 171
Active level	
limit switches	108
Amplifier fault input	
enable	166
Analog I/O	
configuring	156
testing	156
Analog input	
description	154
joystick	105
reporting	157, 210, 327
Analog output	
calibration	158
description	155
max. loading	155
setting	158, 212, 335, 342
API	
components	15
installation	13
Application program samples	44, 45, 46, 47, 48
Application programming	
C++	44
Delphi	46
LabVIEW	47
Visual Basic	45
Arc motion	96

Contour buffer	98
enable	99, 174
on the fly changes	102
specifying	216
Vector acceleration	97
Vector deceleration	97
Vector velocity	97
ASCII command interface	60, 386
At target	
commanding	118, 191, 207, 350
description	118
Auxiliary encoder	
dual loop servo	123
report position	197, 210, 334
servo	123
stepper	123
testing	127
wiring	126
Axis number	389

## B

Background task	
cancel	214
Backlash compensation	
description	127
enable	127, 181
Battery backup connector	232
BF022	
mounting footprint	268
BF100	
mounting footprint	273
BF160	

mounting footprint	277
Binary command interface	60, 382

---

**C**

C++ programming	44
Calibration	
analog module outputs	158
stepper, constant	89
stepper, on power up	70
Capture data	
actual position	136, 179
DAC output	136, 179
following error	136, 179
optimal position	136, 179
Capture encoder index	185
Command set	
functional listing, MCAPI	163
functional listing, MCCL	283
MCAPI	161
MCCL	303
Communication testing	
DCX-AT200	18
Connector	
DCX module pin numbering	22
DCX-AT200	232, 235
DCX-BF022	266, 267
DCX-BF100	271, 272
DCX-BF160	276
DCX-MC200	241
DCX-MC210	246
DCX-MC260	252
DCX-MC400	254
DCX-MC5X0	256
DCX-MF300	258
DCX-MF310	262
Contact Precision MicroControl	iv
Contour buffer	
description	98
tell contour count	98
Cubic spline interpolation	102, 182
Current sink/source	
digital output	149, 223, 227

---

**D**

DAC output	
plotting	50
DCX Architecture	5, 6, 7, 8
DCX command (MCCL)	
description	64, 281, 303
format	65
pausing a command / sequence	67
repeating	66
single stepping	290
terminating a command / sequence	67

DCX controller communications	
ASCII command interface	60, 386
Binary command interface	60, 382
IEEE-488 (GPIB)	59
ISA bus	59, 60
RS-232	59, 60
DCX module	
axis number	389
connector pin numbering	22
DCX system components	
DCX-AT200	6, 149, 154
DCX-BF022	35, 149
DCX-MC400	149
DCX-AT200	
communications testing	18
DCX-controller communications	
IEEE-488 (GPIB)	61
Deceleration	
disable	50
setting	48, 85, 87, 93, 171
Default directory	
MCAPI	13
Default settings	388
Delphi programming	46
Derivative gain	
description	70
retrieving	200
sampling period	78
setting	78, 167
Digital I/O	
AT200, pin out	232
configuring	150, 210
description	149
output, max current	149, 223, 227
testing	150
turn off	152, 211
turn on	152, 211
Direction	
setting	95, 180
Download	
text file	290
Dual Loop servo	123
Dual loop servo control	82
Dual ported memory	
data tables	377
description	377
Dwell	
period of time	188

---

**E**

Encoder	
auxiliary	123
capture index	185
checkout	54, 72
description	70
reversed phased	76

rollover	130
Encoder Index	
checkout	110
description	70
Error codes	
MCAPI	366
MCCL	367
Error LED's	231
E-stop	
enable	128
examples	128
hard wired	128

---

**F**

Fail safe operation	
watchdog circuit	145
Feed forward	82, 120, 145
acceleration	84, 121
calculating	82, 120
deceleration	84, 121
described	82
setting	83, 120
File operations	392
Firmware upgrade	131
Fluid dispensing	
example	292
Following error	
default setting	72
demonstrated	86
description	72
disable	73, 310
plotting	50
setting	167
Formatted messages	292
Friction	85
effects upon system	78
Frictionless servo	
usinf output deadband	309
using output deadband	85

---

**G**

Gearing	
enable	104, 182
setting ratio	104, 182
terminate	104, 182

---

**H**

Home sensor	
checkout	110
Homing an axis	
closed loop stepper	117
encoder index	111, 112, 183, 184, 185
home sensor	112, 183

limit sensor	113
servo	109
stepper	114
troubleshooting	364
HPGL plotting	395

---

**I**

Inertia	
effects upon system	78
Installation	
DCX modules	22
DCX-BF022	35
DCX-MC200	24
DCX-MC210	28
DCX-MC400	35
DCX-MC5X0	36
DCX-MF300	36
DCX-MF310	37
MCAPI	13
Software	13
Integral gain	
description	70
retrieving	200
setting	78, 167
Integral limit	
description	80
setting	80
ISA bus signals	235

---

**J**

Jogging	
description	105
enable	106, 182
setting maximum velocity	106
terminate	106
wiring	105
Joystick controlled motion	105
Jumpering	
analog input (DCX-AT200)	154
DCX-AT200	233, 234
DCX-BF022	268
DCX-MC200	24, 242
DCX-MC210	28, 247
DCX-MC260	32, 253
DCX-MF300	258, 259
DCX-MF310	37, 263

---

**L**

LabVIEW programming	47
Laser cutting	
description	132
example	132
Learning points	135, 186

LED's	
error	231
Limiting the servo command output	141
Limits	
active level	108
checkout	107
disable	107, 172
enable	107, 172
hard (switch / sensor)	107, 172
homing an axis	113
inverting active level	107, 108, 172
normally closed switch	107, 108
programmable	107, 172
troubleshooting	364
Linear interpolation	96
Contour buffer	98
enable	98, 136, 174
on the fly changes	102
specifying	97, 216
Vector acceleration	97
Vector deceleration	97
Vector velocity	97
loading DCX data	
user register	298
Loading motor status	299

---

**M**

Macro command	
as background task	288
defining	286
described	286
execute	214
execution upon reset / power up	287
memory size	286
non volatile	286
reporting	287
resetting (deleting)	287
volatile	286
Master / Slave	
description	104
enable	104
slave ratio	104
tangential knife control	138
termination	104
threading	140
MCAPI	
components	15
default directory	13
installation	13
MCCL commands issued by	88, 132, 138, 140
Setup	16, 64
testing	18
uninstall	21
MCCL commands	
issued via MCAPI	88, 132, 138, 140
not supported by MCAPI	88, 132, 138, 140

Minimum requirements	
PC	5
Module	
communication	7
I/O	7
motion control	6
Motherboard, motion control	5, 6
Motion complete	
at target	118
closed loop stepper	87, 88, 91
description	118
trajectory complete	118
Motion control	
backlash compensation	127
Constant velocity move	95
Contour move	96
laser cutting	132
Learning / Teaching points	135
Master / Slave	104
pause motion	389
Point to point	95
required settings	93
resume motion	389
Tangential knife	138
theory of operation	69
threading	140
Torque mode	141
Motion Integrator	
analog I/O	156
analog output calibration	158
description	49
digital I/O	150
encoder checkout	72
encoder index checkout	110
home sensor checkout	110
<b>limit sensor checkout</b>	107
troubleshooting	355
Motor control output	
DCX-MC200	69
DCX-MC210	69
limiting	141
PWM	132
Motor status	330
loading	299
Mounting footprint	
BF022	268
BF100	273
BF160	277
Moving motors	
required settings	65
Servo motor	70
Stepper motor	87
Multi-tasking	
commands not supported	288
CPU utilization	289
described	288
digital I/O	293
example	289

global data registers-----	289
passing data between-----	289
private data registers-----	289
quantity supported-----	289
termination-----	289
testing-----	288

## N

Normally closed limit switch-----	107, 108
-----------------------------------	----------

## O

OLE Server	
Integrator test panels-----	51
On the fly changes	
arc and linear motion-----	102
Constant velocity motion-----	119
Point to point-----	119
Trapezoidal velocity profile-----	119

## P

Parabolic velocity profile	
description-----	95
Pause motion-----	389
Pausing	
MCCL command / sequence-----	67
PC requirements, minimum-----	5
PDF	
described-----	369
document printing-----	368, 369
viewing a document-----	369
Phasing	
output/encoder-----	72, 76, 91, 175
PID digital filter-----	See Tuning the servo
algorithm-----	70
'D' term-----	70
description-----	70
'I' term-----	70
'I' term	
not used-----	73
'P' term-----	70
rate selection-----	73
theory of operation-----	70
Pin out	
DCX module connector-----	22
DCX-AT200 gen. purpose I/O-----	232
DCX-BF022-----	266, 267
DCX-BF100-----	271, 272
DCX-BF160-----	276
DCX-MC200-----	241
DCX-MC210-----	246
DCX-MC260-----	252
DCX-MC400-----	254
DCX-MC5X0-----	256

DCX-MF300-----	258
DCX-MF310-----	262
PLC	
analog I/O-----	296
digital I/O-----	292
Plotter file (HPGL)-----	395
PMC email address-----	iv
PMC web address-----	iv
Point to point motion	
execution-----	95, 186, 187
Position	
Recording-----	136
redefining-----	173
Position mode	
enable-----	95, 174
Printing a PDF document-----	368, 369
Program samples-----	44, 45, 46, 47, 48
Proportional gain	
description-----	70
retireving-----	200
setting-----	76, 167
PWM servo control-----	132
enable-----	173

## R

Recording position data-----	136
Remote vehicle control	
example-----	296
Repeating	
command or sequence-----	66
Report	
analog input value-----	210
axis 'at target'-----	118, 207
captured data-----	136, 198
current position-----	202
Digital channel-----	211
following error-----	200
MCAPI errors-----	196
motor command output-----	206
motor status-----	195, 205, 330
optimal position-----	202
PID parameters-----	200
programmed velocity-----	207
scaling factors-----	204
target position-----	206
trajectory complete-----	118, 208
user register-----	203
Reporting	
firmware version-----	20
MCAPI version-----	20
Reset	
manual (external switch)-----	137
software-----	187
Resume motion-----	389
Rollover	
encoder-----	130

**S**

Sales support .....	iv
Scaling	
defining user units .....	143, 169
S-curve velocity profile	
description .....	95
Servo command output	
+/- 10V .....	173
0V - +10V .....	173
limiting .....	141, 176
PWM .....	173
Servo loop	
description .....	70
Servo loop rate	
selection .....	73
Servo motor control	
homing .....	109
theory of operation .....	69
tuning the servo .....	74
Setup	
MCAPI .....	16, 64
Single stepping a program .....	290
Software	
default directory .....	13
Demo programs .....	55
Flash Wizard .....	53
installation .....	13, 15
Motion Integrator .....	49, 107, 150, 156, 355
New Controller Wizard .....	16
OLE Server .....	51
On-line help .....	55
Position Readout .....	54
reporting firmware version .....	20
reporting software version .....	20
Servo Tuning utility .....	74
setup .....	52
source code .....	55
uninstall MCAPI .....	21
WinControl .....	290, 367
Specifications	
DCX-AT200 .....	223
DCX-MC200 .....	69, 222, 225
DCX-MC210 .....	69, 222, 225
DCX-MC260 .....	222, 226
DCX-MC400 .....	227, 254
DCX-MC5X0 .....	222, 229, 256
DCX-MF300 .....	229
DCX-MF310 .....	229
Stand-alone applications .....	391
Status	
motor .....	205, 330
Status LED's .....	231
Step rate	
selection .....	87, 226
Stepper motor	
homing .....	117
Stepper motor control	

closed loop .....	88
homing .....	114
open loop .....	87
output, CW / CCW .....	173
output, Pulse & Dir. ....	173
theory of operation .....	70
Stop move .....	178, 188
Switch settings	
DCX-AT200 .....	12, 234
DCX-MF310 .....	38, 264

**T**

Tangential knife control	
description .....	138
example .....	138
Teaching points .....	135
Technical support .....	iv
Terminating	
MCCL command / sequence .....	67
Testing	
analog I/O .....	156
DCX-AT200 communication .....	18
digital I/O .....	150
MCAPI .....	18
Text file	
download .....	290
Threading operations	
description .....	140
Timeout, MCAPI	
setting .....	219
Trajectory complete	
closed loop stepper .....	87, 88, 91
description .....	118
Trajectory generator	
demonstrated .....	86
description .....	69
disable .....	76
disable (Gain mode) .....	174
enable .....	86
Trapezoidal velocity profile	
description .....	95
Troubleshooting	
encoder checkout .....	54, 72
general .....	356
home sensor input .....	364
limit switches .....	364
'PC' bus communication .....	357
servo motion .....	359, 361
servo tuning .....	358
stepper motion .....	362
Tuning the servo	
derivative gain .....	78
derivative sampling period .....	78
description .....	74
high inertia systems .....	78
initial settings .....	76



integral gain -----	78
integral limit-----	80
proportional gain-----	76
range of slide controls-----	81
restoring settings-----	81
saving settings-----	81, 84
Servo tuning utility -----	74
Velocity mode amplifier-----	81

---

**U**

Uninstall	
MCAPI -----	21
Upgrade	
firmware -----	131
User memory-----	302
User registers	
description -----	298
User units	
controller time base-----	144
description -----	143
machine zero-----	144
output constant -----	145
part zero-----	144
setting-----	143, 169
trajectory time -----	144
user scale -----	143, 144

---

**V**

Vector acceleration -----	97, 168
Vector deceleration-----	97, 168
Vector velocity -----	97, 168, 177
Velocity	
disable -----	50, 76
set too high -----	72
setting-----	48, 85, 176
Velocity gain-----	89, 145
Velocity mode	
enable-----	95
Velocity mode amplifier	

description -----	81, 120
tuning -----	81
Velocity mode move	
enable -----	174
execution-----	95
setting direction-----	180
setting the direction-----	95
starting-----	96, 185
Velocity profiles	
Contour mode motion-----	96
Parabolic-----	69, 95, 174
S-curve -----	69, 95, 174
Trapezoidal-----	69, 95, 174
Version	
firmware-----	20
software -----	20
Visual Basic programming-----	45

---

**W**

Wait	
for absolute position-----	189
for 'at target' -----	118, 191
for coarse home sensor -----	189
for digital channel =-----	212
for relative position -----	190
for trajectory complete -----	118, 190
period of time -----	188
Watchdog circuit	
description -----	145
Wiring	
auxiliary encoder-----	126
axis I/O -----	26, 27, 30, 31, 33, 34
encoder, reversed phased-----	76
E-stop-----	128
Joystick-----	105
manual reset switch -----	137
servo amplifier -----	26, 27
servo axis -----	26, 27, 30, 31
stepper axis -----	33, 34
stepper driver -----	33, 34