EPCIO Series

Motion Control Command Library

User Manual

(Applicable to Motion Control Command Library V.5.10)

Version: V. 5.10

Date: 2009.10

http://www.epcio.com.tw

Table of Contents

1. Introduction to the Motion Control Command Library

The EPCIO Series Motion Control Command Library (MCCL) provided with the EPCIO Series motion control cards can be used in WINDOWS 98SE and WINDOWS NT/2000/XP work platforms, and supports Visual C++, Visual Basic, and Borland C++ Builder development environments.

The MCCL provides spatial trajectory planning commands such as point-to-point, straight line, curve, circular, and helix motions. Additionally, the MCCL also provides 16 types of the go home modes, motion dry run, motion delay, pulse motion /inch motion/continuous inch motion, and pause, continue, and abort motions.

For the trajectory planning function, settings include different acceleration/deceleration times, acceleration/deceleration curve types, feed speeds, maximum feed speed, and maximum acceleration. The MCCL also includes software and hardware over-travel protection, path blending, dynamic feed speed adjustments, and error information handling.

For the in position controls, the user can use the MCCL to set the in position proportional gain and error tolerance. The MCCL also provides the in position confirmation, gear backlash, and gap compensation.

For handling the I/O connection signal, the user can utilize the MCCL to acquire the Home connection and Limit Switch connection signals, and to output the Servo-On/Off signal. Additionally, certain I/O input signals automatically trigger the interrupt service routine (ISR), but the user can customize the ISR.

For the encoder, the user can promptly acquire the encoder count and set the encoder signal input rate. Certain input signals automatically latch the encoder count. The MCCL supports a function which triggers the user-customized ISR when the encoder count reaches a given value.

For D/A conversion, users can not only use the MCCL to output the required external voltage (-10 V to 10 V), but can also preprogram the desired input voltage, and automatically output this voltage once the trigger conditions have been satisfied.

For A/D conversion, the user can use the MCCL to acquire the input voltage (-5 V to 5 V or 0 V to 10 V; -10 V to 10 V or 0 V to 20 V) and set unitary or label channel voltage conversion. The ISR is triggered once the conversion work is complete or the voltage satisfies the comparative condition. The user can customize the ISR.

For the timer, the user can set the time limit. Once the timer is enabled and the time ends, the user-customized ISR is automatically triggered and the timer is reset. This process continues until the function is disabled. The MCCL also provides a Watchdog function.

Using the MCCL does not require an in-depth understanding of the complex trajectory planning, in position control, and real-time multi-tasking environment. With this command library, users can quickly develop and integrate systems.

Related Reference Manuals:

Hardware Information

EPCIO – 400/405 Hardware User Manual

EPCIO – 601/605 Hardware User Manual

EPCIO – 4000/4005 Hardware User Manual

EPCIO – 6000/6005 Hardware User Manual

Motion Control Command Library User Guides

EPCIO Series Motion Control Command Library Reference Manual

EPCIO Series Motion Control Command Library Examples Manual

EPCIO Series Motion Control Command Library Integrated Testing Enviroment Manual

2. MCCL Functions

2.1 Software Specifications

■ **Work System Environment**

✓ WINDOWS 98SE

✓ WINDOWS NT

✓ WINDOWS 2000

✓ WINDOWS XP/XP Embedded

■ Development Environment

✓ Borland C++ Builder (BCB)

✓Visual C++ (VC++)

✓ Visual Basic (VB)

Required Files When Using the MCCL

	PCI	**ISA**
VC++	MCCL.h MCCL_Fun.h MCCLPCI_50.lib	MCCL.h MCCL_Fun.h MCCLISA_50.lib
BCB	MCCL.h MCCL_Fun.h MCCLPCI_50_BCB.lib	MCCL.h MCCL_Fun.h MCCLISA_50_BCB.lib
VB	MCCLPCI_50.bas	MCCLISA_50.bas

2.2 Motion Axis Definitions and the Maximum Number of Combinable Control Cards

2.2.1 Motion Axis Definition

The purpose of MCCL design is to provide motion control functions for three orthogonal axes (XYZ) with three auxiliary axes (UVW). See Fig. 2.2.1. U, V, and W are the three auxiliary axes representing three independent axial directions.

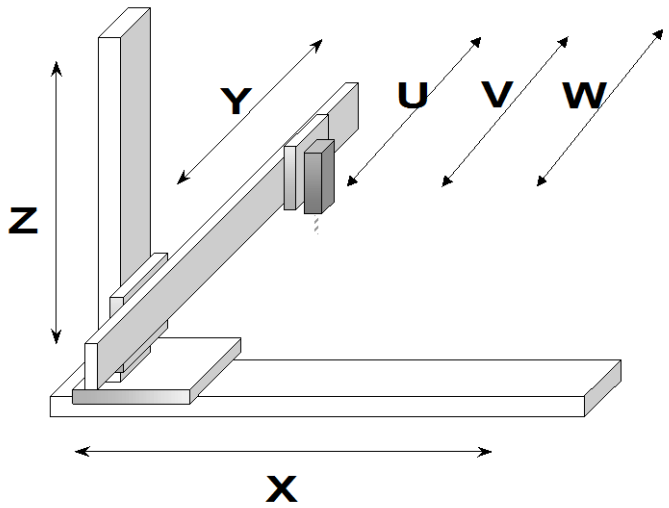


Fig. 2.2.1. Three orthogonal axes (**XYZ**) with three auxiliary axes (**UVW**)

The MCCL provides a maximum of 6 control axes with simultaneous movement. The user can utilize an EPCIO Series motion control card to simultaneously or separately control 1 to 6 axes. The user can select absolute or incremental positions for given motion commands. This command library will record the motion's absolute position (corresponding to the home position) regardless of the position type selected.

2.2.2 Maximum Number of Combinable Control Cards

Each motion control card can control up to a six groups of system (both motor and driver) (EPCIO-601/605/6000/6005) or four groups of system (EPCIO-400/405/4000/4005), depending on type. The motion control command library can

control up to 12 motion control cards simultaneously, thereby controlling a maximum of 72 axes. EPCIO Series motion control cards can send velocity commands (only EPCIO-400/601/4000/6000) or pulse commands (all products in the EPCIO series). The basic configuration is displayed in Fig. 2.2.2.

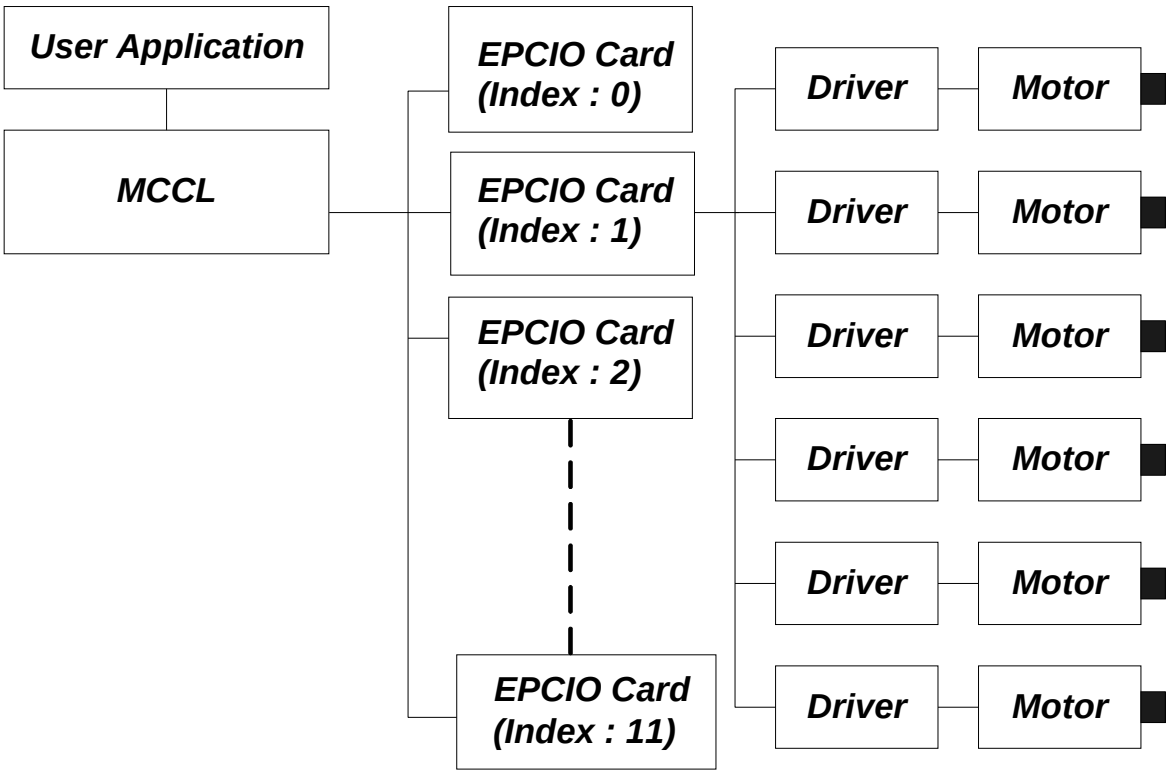


Fig. 2.2.2. The MCCL can control 12 EPCIO series motion control cards

2.3 Command Library Operational Properties

After motion commands are called in the MCCL, the related motion commands will first be put into each group's exclusive motion command queue and ***will not be executed immediately*** (for a description of groups, please refer to **2.5.1 Initiating the Motion Control Command Library).** The MCCL will then use the first in first out (FIFO) method to get the motion command from the queue to conduct interpretation and caculate the interpolation (refer to Fig. 2.3.2). Putting and getting the commands are non-sequential and asynchronized actions. It is unnecessary to wait for the completion of one motion command before a new motion command can be sent to the motion command queue.

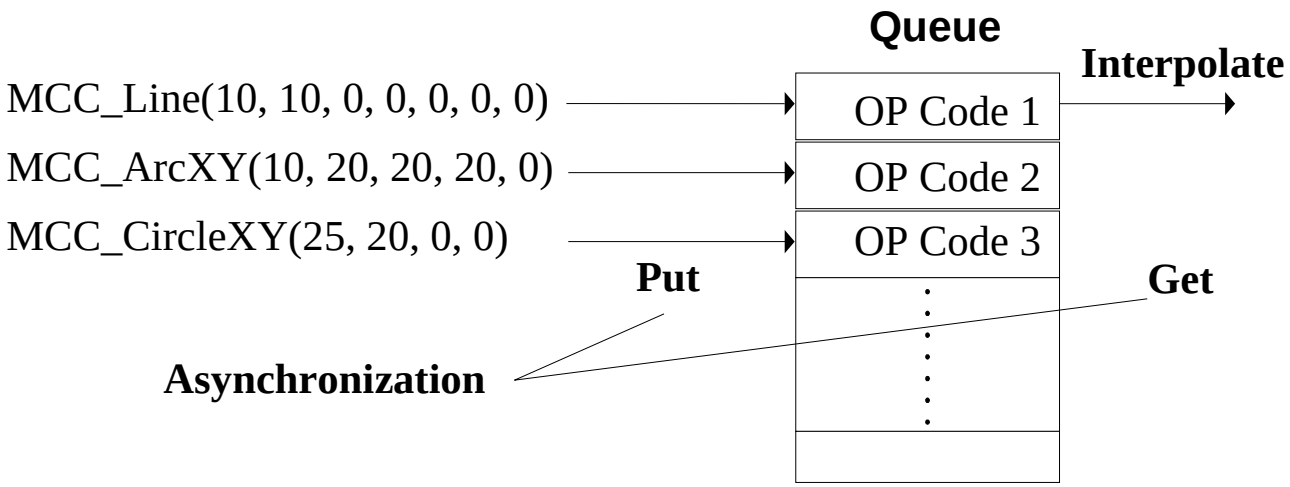


Fig. 2.3.2 Motion Command Queue

The motion command queue for each group is preset to store 10,000 commands (10,000 is the maximum), but MCC_SetCmdQueueSize can be used to change the size, and MCC_GetCmdQueueSize can be used to acquire the current size of the queue. Notably, ***the queue size can only be changed when the queue is empty.***

The following is a list of command names that will increase the motion command library storage capacity.

By calling these commands, the MCCL will put a command in the motion command queue, get the first command in the queue at the appropriate time, and perform the corresponding action.

A. Straight Line Motion Command

MCC_Line()

B. Curved Motion Commands

MCC_ArcXYZ() MCC_ArcXYZUVW()

MCC_ArcXY() MCC_ArcYZ()

MCC_ArcZX()

MCC_ArcXYUVW() MCC_ArcYZUVW() MCC_ArcZXUVW()

MCC_ArcThetaXY() MCC_ArcThetaYZ() MCC_ArcThetaZX()

C. Circular Motion Commands

MCC_CircleXY() MCC_CircleYZ() MCC_CircleZX()

MCC_CircleXYUVW() MCC_CircleYZUVW() MCC_CircleZXUVW()

D. Helix Motion Commands

MCC_HelicaXY_Z() MCC_HelicaYZ_X() MCC_HelicaZX_Y()

E. Point-to-Point Motion Command

MCC_PtP()

F. Inch Motion and Continuous Inch Motion Commands

MCC_JogSpace() MCC_JogConti()

G. Confirm In Position Commands

MCC_EnableInPos() MCC_DisableInPos()

H. Path Blending Commands

MCC_EnableBlend() MCC_DisableBlend()

I. Delay Motion Command

MCC_DelayMotion()

Using the above commands when the motion command queue is already full will result in a return value of COMMAND_BUFFER_FULL_ERR, meaning that the command cannot be accepted. Figure 2.3.2 shows the operational process for the Group 0 motion command queue, and demonstrates that commands belonging to the same group will be executed sequentially. Because each group has an exclusive motion command queue, motion commands from different groups can be executed simultaneously.

CAUTION:

If you have the requirement: After making the Group 0 X axis moves to the position at coordinate 10, output the servo-on signal and further move the axis to coordinate 20. The program could be written like this:

```
MCC_Line(10, 0, 0, 0, 0, 0, 0);
MCC_SetServoOn(1, 0);
MCC_Line(20, 0, 0, 0, 0, 0, 0);
```

Then, once MCC_Line() has been placed in the motion command queue (but has yet to be executed), immediately perform MCC_SetServoOn(). Because MCC_SetServoOn() is not placed in the queue but instead directly executed, the servo-on signal will have already been sent before the actual position reaches coordinate 10. This operational property requires special attention.

If the signal output needs to be executed after the X axis reaches coordinate 10, additional user determination is required. Users must inspect the system motion status or current position to control the signal output. Below is a simple example:

```
// Assume the X axis in Group 0 is required to move to coordinate 10 before servo-on is output
// signal
MCC_Line(10, 0, 0, 0, 0, 0, 0);

while (MCC_GetMotionStatus(0) != GMS_STOP)
```

```
// MCC_GetMotionStatus() return value equaling GMS_STOP indicates that currently all
// motion commands have been completed
MCC_SetServoOn(1, 0);
MCC_Line(20, 0, 0, 0, 0, 0, 0);
```

2.4 Machine, Encoder, and Go Home Parameter Settings

2.4.1 Machine Parameters

The MCCL uses machine parameters to define the user's machine platform characteristics and driver usage type by using a logical home positioning system, position system boundary values, or maximum safe feed speeds for each axis corresponding to machine parameter planning.

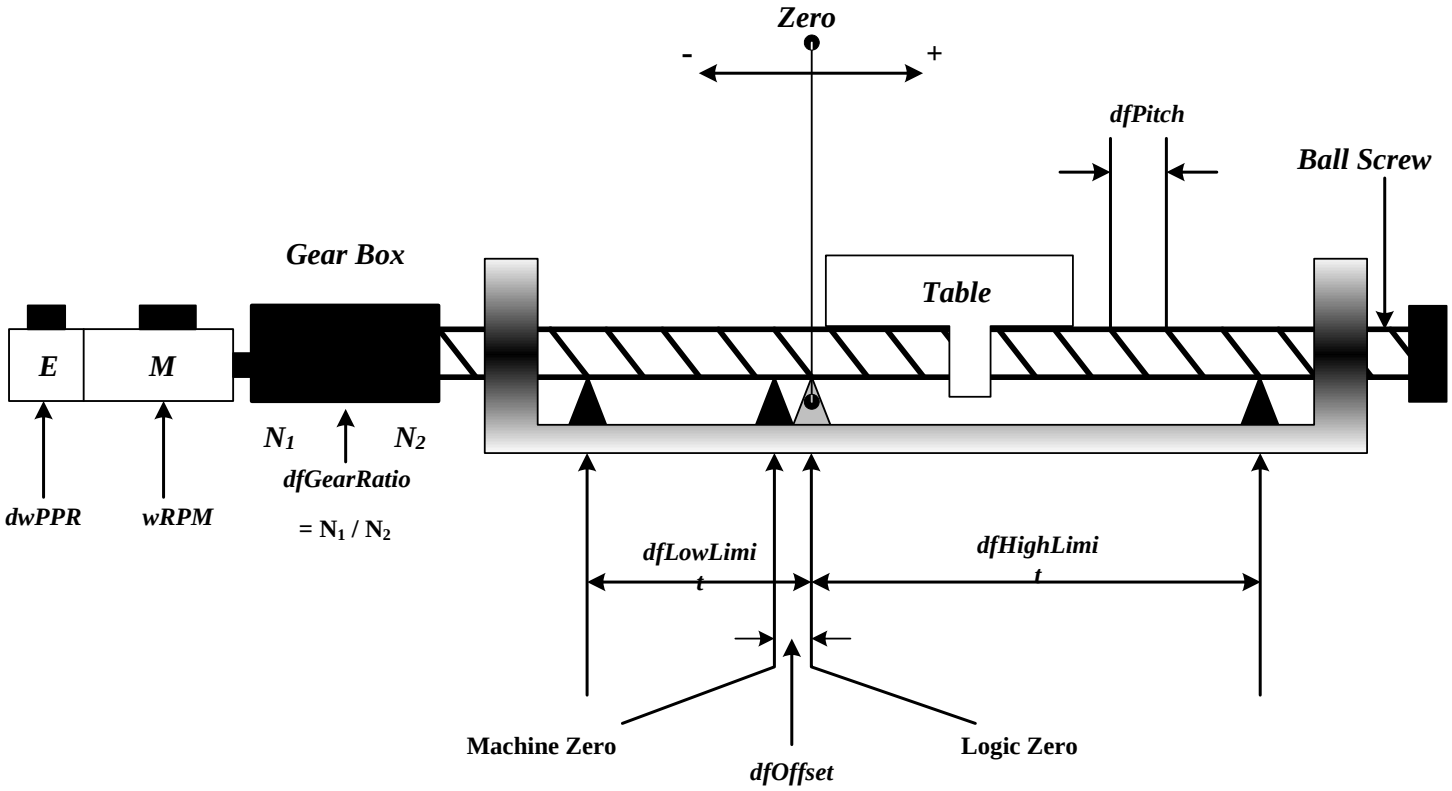


Fig. 2.4.1. Structural platform charcteristics

Below is a detailed description of the machine parameters:

typedef struct *_SYS_MAC_PARAM*

{

WORD *wPosToEncoderDir*;

WORD *wRPM*;

DWORD *dwPPR*;

```
    double      dfPitch;
    double      dfGearRatio;
    double      dfHighLimit;
    double      dfLowLimit;
    double      dfHighLimitOffset;
    double      dfLowLimitOffset;
    WORD        wPulseMode;
    WORD        wPulseWidth;
    WORD        wCommandMode;
    WORD        wPaddle;
    WORD        wOverTravelUpSensorMode;
    WORD        wOverTravelDownSensorModee;
} SYS_MAC_PARAM;
```

***wPosToEncoderDir*:** Directional adjustment parameter

0 Output command does not reverse

1 Output command reverses

This parameter revises the direction of the motion command when it differs from the desired structural motion direction. If a forward direction motion command such as MCC_JogSpace(10, 10, 0, 0) is sent, but due to motor wiring the structure actually moves in the direction opposite the user's definition, this parameter could be set to "1" to align the directions of the motion command with the desired direction of the structural motion (altering motor wiring is not required).

***dwRPM*:** Maximum number of safe motor rotations per minute for each axis

When conducting fast point-to-point motion, each axis's number of rotations per minute, which is converted from the speed setting, will not exceed the set value for *dwRPM*.

➔ *See Also* MCC_SetPtPSpeed()

wPPR: Increase in the encoder count for each revolution of the motor shaft or number of pulses required per rotation

If closed circuit control is used, this value is the increase in the encoder count for each revolution of the motor shaft; if it is an open circuit system, the value is the number of pulses required per revolution.

When using a linear motor, *dfPitch* and *dfGearRatio* should both be set to 1. Additionally, a linear motor has no definition related to *wPPR* and the distance required to move is often calculated in terms of pulses, so *wPPR* can be set to 1 and the units used in the MCCL can be changed to pulse. For example, when the X axis needs to move 1000 pulses, MCC_Line(1000, 0, 0, 0, 0, 0) can be called for the X axis to output 1000 pulses. Using MCC_SetFeedSpeed(500) means that required linear motor speed is 500 pulses per minute.

***dfPitch*: Ball screw backlash**

The distance the table moves for each revolution of the ball screw. Units: mm or inch. If there is no ball screw configuration, this value should be set to 1.

***dfGearRatio*:** Gearbox deceleration ratio

The two-way gear ratio for the gearbox connecting the motor shaft and the ball screw can be calculated using the number of gear gaps, or is simply defined as "number of motor rotations per ball screw revolution." If the gearbox is not configured, this value should be set to 1.

dfHighLimit: Positive boundary for over travel software (also called high limit)

This value is the maximum positive displacement allowed from the logical home position. Units: mm or inch.

➔ *See Also* MCC_SetOverTravelCheck()

***dfLowLimit*:** Negative boundary for over travel software (also called low limit)

This value is the maximum negative displacement allowed from the logical home position and is often set as a negative value. Units: mm or inch.

dfHighLimitOffset:

To preserve the field, set to 0.

dfLowLimitOffset:

To preserve the field, set to 0.

wPulseMode: Pulse output mode

DDA_FMT_PD	Pulse/Direction
DDA_FMT_CW	CW/CCW
DDA_FMT_AB	A/B phase

wPulseWidth: Pulse output width

The length of the pulse output width should conform to the required drive specifications. Actual output pulse width is equal to this set value multiplied by the system cycle time (25 ns). Pulse output width should be set according to the required drive specifications. Normally, this value should not be less than 40.

wCommandMode: Motion command output mode

OCM_PULSE	Pulse Command
OCM_VOLTAGE	Velocity Command

Caution: ***wPulseMode*** and ***wPulseWidth*** only have meaning when this value is OCM_PULSE.

wPaddle

To preserve the field, set to 0.

wOverTravelUpSensorMode: Positive Limit switch wiring; please refer to the below description of how to verify that the wiring is correct.

SL_NORMAL_OPEN	Active High
SL_NORMAL_CLOSE	Active Low
SL_UNUSED	Does not check if the limit switch has been triggered. This item can be used if the limit switch has yet to be installed for the axis indicated.

wOverTravelUpSensorMode: Negative Limit switch wiring; please refer to the below description of how to verify that the wiring is correct.

SL_NORMAL_OPEN	Active High
SL_NORMAL_CLOSE	Active Low
SL_UNUSED	Does not check if the limit switch has been triggered. This item can be used if the limit switch has yet to be installed for the axis indicated.

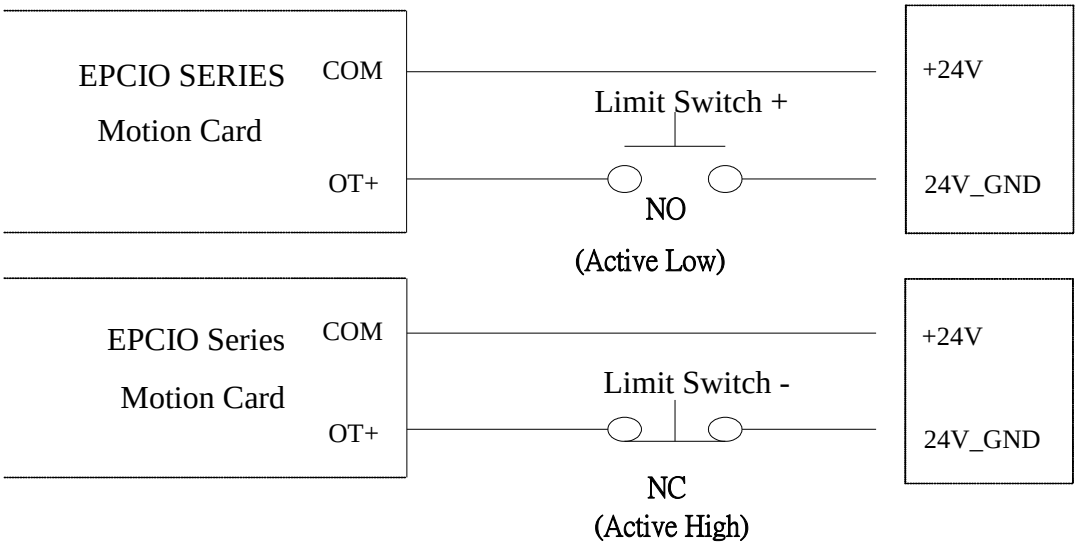


Fig. 2.4.2. Limit switch wiring

To use the limit switch function, *wOverTravelUpSensorMode* and *wOverTravelDownSensorMode* must be accurately set according to the limit switch wiring (see Fig. 2.4.2). MCC_GetLimitSwitchStatus() can be used to verify the accuracy of the wiring settings. If the limit switch status obtained by MCC_GetLimitSwitchStatus() is active when the limit switch has yet to be triggered, then there is an error in the wiring settings, and *wOverTravelUpSensorMode* or *wOverTravelDownSensorMode* needs to be reset.

For the limit switch to operate normally, in addition to accurately setting the limit switch wiring, MCC_EnableLimitSwitchCheck() must also be called for the

wOverTravelUpSensorMode and *wOverTravelDownSensorMode* settings to be effective.

However, calling MCC_EnableLimitSwitchCheck() is meaningless if *wOverTravelUpSensorMode* and *wOverTravelDownSensorMode* are set to SL_UNUSED.

When this function is enabled, triggering the limit switch for the direction of the given axis (for example, triggering the positive limit switch when moving in a positive direction, or triggering the negative limit switch when moving in a negative direction) will stop the output group motion command (and produce an error record).

MCC_EnableLimitSwitchCheck() is often used in combination with MCC_GetErrorCode(). Continuously calling MCC_GetErrorCode () verifies whether the system has produced an error record by triggering a limit switch (codes 0xF701 to 0xF706 represent triggered limit switches for axes X to W, respectively). When an error from a triggered limit switch is discovered, a message will display on the screen, alerting the operator. MCC_ClearError() is then called during programming to clear the error and allow the system to travel in the opposite direction, away from the limit switch.

After each field content within the machine parameters is confirmed, the machine parameters can be set using MCC_SetMacParam(). Below is an example of this command:

```
SYS_MAC_PARAM     stAxisParam;
memset(&stAxisParam, 0, sizeof(SYS_MAC_PARAM)); // clear content to zero

stAxisParam.wPosToEncoderDir                = 0;
stAxisParam.dwPPR                           = 500;
stAxisParam.wRPM                            = 3000;
stAxisParam.dfPitch                         = 1.0;
stAxisParam.dfGearRatio                     = 1.0;
stAxisParam.dfHighLimit                     = 50000.0;
stAxisParam.dfLowLimit                      = -50000.0;
stAxisParam.wPulseMode                      = DDA_FMT_PD;
```

```
stAxisParam.wPulseWidth                 = 100;
stAxisParam.wCommandMode                = OCM_PULSE;
stAxisParam.wOverTravelUpSensorMode     = SL_UNUSED; // not check
stAxisParam.wOverTravelDownSensorMode   = SL_UNUSED; // not check

MCC_SetMacParam(&stAxisParam, 0, 0); // set axis 0 in card 0
```

The machine parameters must be set before the MCC_InitSystem() is used. Additionally, the machine parameters in various axes must be set separately.

→*See Also* MCC_GetMacParam()

2.4.2 Encoder Parameters

The MCCL uses encoder parameters to define the encoder characteristics, including the encoder signal input types, signal input phase swap status, and feedback rates (x 1, x 2, and x 4). A detailed description of the encoder parameters is provided below:

```
typedef struct _SYS_ENCODER_CONFIG
{
    WORD        wType;
    WORD        wAInverse;
    WORD        wBInverse;
    WORD        wCInverse;
    WORD        wABSwap;
    WORD        wInputRate;
    WORD        wPaddle [2];
} SYS_ENCODER_CONFIG;
```

wType: Input type setting

ENC_TYPE_AB A/B Phase

ENC_TYPE_CW CW/CCW

ENC_TYPE_PD Pulse / Direction

wAInverse: Phase A signal inverse status

0 Not inverse

1 Inverse

wBInverse: Phase B signal inverse status

0 Not inverse

1 Inverse

wCInverse: Phase C (Phase Z) signal inverse status

0 Not inverse

1 Inverse

wABSwap: Phase A/B signal swap status

0 Not swap

1 Swap

wInputRate: Set encoder feedback rate

1 Feedback rate of 1 (×1)

2 Feedback rate of 2 (×2)

4 Feedback rate of 4 (×4)

paddle: To preserve the field, set to 0.

After each field content within the encoder parameters is confirmed, the encoder parameters can be set using MCC_SetEncoderConfig(). Below is an example of this command:

```
SYS_ENCODER_CONFIG   stENCConfig;
memset(&stENCConfig, 0, sizeof(SYS_ENCODER_CONFIG));

stENCConfig.wType          = ENC_TYPE_AB;
stENCConfig.wAInverse      = 0; // not inverse
```

```
stENCConfig.wBInverse      = 0; // not inverse
stENCConfig.wCInverse      = 0; // not inverse
stENCConfig.wABSwap        = 0; // not swap
stENCConfig.wInputRate     = 4; // set encoder input rate: x4

MCC_SetEncoderConfig(&stENCConfig, 0, 0); // set axis 0 in card 0
```

Before using MCC_InitSystem(), the encoder parameters of each axis must be set separately.

CAUTION

If the machine or encoder parameters are altered after MCC_InitSystem() has been called, MCC_UpdateParam() must also be called for the system to be able to respond to the new settings. ***However, the effect of using MCC_UpdateParam() is similar to that when using MCC_ResetMotion(): the system will reset to the initial status created after MCC_InitSystem() is called.***

2.4.3 Go Home Parameters

The MCCL uses the Go Home parameters to define the Go Home action, including mode of use, direction of Go Home motion, Home sensor wiring, encoder index signal counts, and acceleration/deceleration settings. For a more detailed explanation, please read "**2.8 Go Home.**"

The Go Home parameter content is described below:

```
typedef struct _SYS_HOME_CONFIG
{
    WORD        wMode;
    WORD        wDirection;
    WORD        wSensorMode;
    WORD        wPaddel0;
    int         nIndexCount;
    int         nPaddel1;
```

double *dfAccTime*;
double *dfDecTime*;
double *dfHighSpeed* ;
double *dfLowSpeed* ;
double *dfOffset*;
} *SYS_HOME_CONFIG*;

wMode: Go Home mode

This defines the Go Home mode used. This parameter value must be greater than 3 and less than 16. For a detailed description of each mode, please consult the section related to Go Home.

wDirection: Initial direction of Go Home motion

0 Positive

1 Negative

wSensorMode: Home sensor wiring

SL_NORMAL_OPEN Active High

SL_NORMAL_CLOSE Active Low

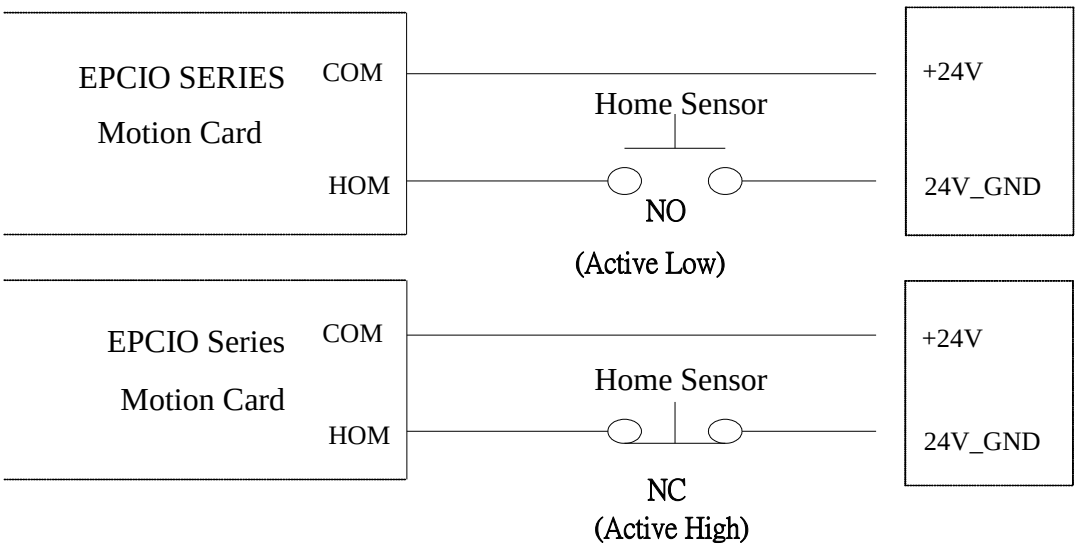


Fig. 2.4.3. Home sensor wiring

To use the Go Home function, *wSensorMode* must be accurately set according to the home sensor wiring (see Fig. 2.4.3). MCC_GetHomeSensorStatus() can be used to verify the accuracy of the wiring settings. If the home sensor status obtained by MCC_GetLimitSwitchStatus() is active despite the home sensor not having been triggered, then there is an error in the wiring, and the *wSensorMode* setting needs to be altered.

nIndexCount: Indicated encoder index signal code
For phase 2 in the Go Home motion process (seeking indicated code index), the code for the first index signal that occurs is 0, the code for the second index signal that occurs is 1, and subsequent index signals follow this pattern. Some Go Home modes require the indicated encoder index signal code to be able to complete the entire Go Home motion once the signal has been triggered.

dfAccTime: The time required to accelerate to *dfHighSpeed* or *dfLowSpeed* during the Go Home motion. Unit: ms

dfDecTime: The time required to decelerate from *dfHighSpeed* or *dfLowSpeed* to a stop during the Go Home motion. Unit: ms

dfHighSpeed: High speed setting. Unit: mm/s or inch/s.
This command is often the speed used during the first phase of the Go Home motion.

dfLowSpeed: Low speed setting. Unit: mm/s or inch/s.
This command is often the speed used during the final phase of completing the Go Home motion.

dfOffset: Distance from the logical home position. Unit: mm or inch
Generally, the displacement between the machine home and the logical home will be found during inspection. To confirm this displacement, first set *dfOffset* to 0. When the Go Home action is complete (the platform has stopped at the “machine home”), use the JOG driver method to find the displacement from the “logical home,” and use

this displacement for the *dfOffset* setting. After Go Home has been performed again, the motion axis will move to the "logical home" position and the system will use this point as the motion command reference home.

After each field content within the Go Home parameters is confirmed, the Go Home parameters can be set using MCC_SetHomeConfig(). Below is an example of this command:

```
SYS_HOME_CONFIG       stHomeConfig;
memset(&stHomeConfig, 0, sizeof(SYS_HOME_CONFIG));

stHomeConfig.wMode              = 3;      //usage mode 3
stHomeConfig.wDirection         = 1;      // Go Home motion in a negative direction
stHomeConfig.wSensorMode        = 0;      // use Active High wiring
stHomeConfig.nIndexCount        = 2;      // index code 2
stHomeConfig.dfAccTime          = 300;    // time required for acceleration, units: ms
stHomeConfig.dfDecTime          = 300;    // time required for deceleration, units: ms

stHomeConfig.dfHighSpeed        = 10;     // unit: mm/s or inch/s
stHomeConfig.dfLowSpeed         = 2;      // unit: mm/s or inch/s
stHomeConfig.dfOffset           = 0;

MCC_SetHomeConfig(&stHomeConfig, 0, 0);     // sets axis 0 in card 0
```

The Go Home parameters of each axis must be set separately before the Go Home motion can be executed.

2.4.4 Group (Motion Group) Parameter Setting

All required groups (motion groups) must be established before using the MCCL. A group can be considered an independent motion system. An interdependent relationship often exists between each motion axis within this system during motion. An obvious example is the XYZ table.

For the group operation concept used in the MCCL, the motion control commands provided are primarily operated in units of groups. Each group contains six motion axes: X, Y, Z, U, V, and W; but each motion axis is not required to actually correspond to an output channel on the EPCIO Series motion control card. The MCCL can simultaneously control 12 EPCIO Series motion control cards. Each card has a maximum of 6 defined groups, so the MCCL can simultaneously use up to 72 mutually independent groups without impacting the operations of other groups.

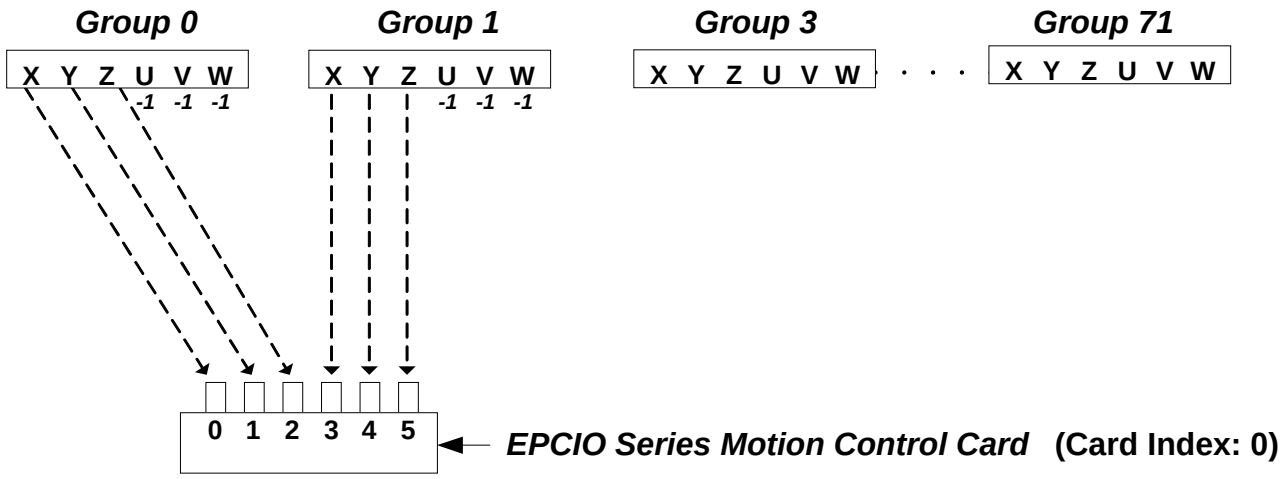


Fig. 2.4.5. Group parameter settings

For example, two groups and one EPCIO Series motion control card are being used in Fig. 2.4.5. The result of the trajectory planning for axes X, Y, and Z in Group(0) are from physical output channels 0, 1, and 2, respectively in Card 0; trajectory planning for axes U, V, and W is ignored. The result of the trajectory planning for axes X, Y, and Z in Group(1) are from physical output channels 3, 4, and 5, respectively in Card 0; trajectory planning for axes U, V, and W is ignored.
Here, the programming can be written as follows:

```
int     nGroup0, nGroup1;

MCC_CloseAllGroups();                  // disable all the groups
nGroup0 = MCC_CreateGroup( 0,        // X corresponds to physical output channel 0
```

```
                                1,      // Y corresp onds to physical output channel 1
                                2,      // Z corresponds to physical output channel 2
                                -1,     // U does not correspond to an physical output channel
                                -1,     // V does not correspond to an physical output channel
                                -1,     // W does not correspond to an physical output channel
                                0);     // corresponds to Card 0

nGroup1 = MCC_CreateGroup( 3,   // X corresponds to physical output channel 3
                           4,   // Y corresponds to physical output channel Channel 4
                           5,   // Z corresponds to physical output channel Channel 5
                           -1,  // U does not correspond to an physical output channel
                           -1,  // V does not correspond to an physical output channel
                           -1,  // W does not correspond to an physical output channel
                           0);  // corresponds to Card 0
```

The return value for MCC_CreateGroup() represents the newly established group code (0 to 71). This code will be used later when calling motion commands. For example, to move the X, Y, and Z axes in Group(1) to coordinate position 10, the program MCC_Line(10, 10, 10, 0, 0, 0, ***nGroup1***) should be written. EPCIO series motion control Cards 0 channels 3, 4, and 5 will be responsible for the interpolation output of axes X, Y, and Z in this group. This is further illustrated in the following example.

```
MCC_Line(10, 10, 10, 0, 0, 0, nGroup0 );    // command 0
MCC_Line(20, 20, 20, 0, 0, 0, nGroup0 );    // command 1
MCC_Line(10, 10, 10, 0, 0, 0, nGroup1 );    // command 2
MCC_Line(20, 20, 20, 0, 0, 0, nGroup1 );    // command 3
```

Using the above group settings, Group(0) will execute command 0 and will output the trajectory planning for axes X, Y, and Z from Card 0 channels 0, 1, and 2. After command 0 is complete, Group(0) can then execute command 1 from the same group.

Because each group operates autonomously, Group(1) is not required to wait for Group(0) to complete command 0 before directly executing command 2, and outputting trajectory planning for axes X, Y, and Z from Card 0 channels 3, 4, and 5. After command 2 is complete, Group(1) can then execute command 3 from the same group.

If no groups have been created before initiating the MCCL, the MCCL will use the default operations. Default operations simply enable the index to be Group(0) and its motion axes X, Y, Z, U, V, and W to correspond to Card 0 channels 0 through 5.

NOTE

1. Groups do not affect each other.
2. Groups all contain six motion axes, X, Y, Z, U, V, and W, that may or may not correspond to a physical channel output. However, **at least one motion axis in the group is required to correspond to a physical channel**. In addition, **two motion axes cannot correspond to the same physical channel.**
3. To reduce CPU usage rate, **minimize the number of groups used.**

2.5 Enabaling and Disabling the Motion Control Command Library

2.5.1 Enabling the Motion Control Command Library

The following parameters must be set prior to using the MCCL:

a. Machine Parameters Use: MCC_SetMacParam()
b. Encoder Parameters Use: MCC_SetEncoderConfig()
c. Group Parameters Use: MCC_CreateGroup() /
MCC_CloseAllGroups()

If these parameters have not been set or if errors occur during these procedures, other commands in the MCCL cannot be used. For machine, encoder, and group (motion group) parameter settings, please read the description in the earlier section of this manual and consult the "**EPCIO Series Motion Control Command Library Examples Manual**." The following only describes how to initiate the MCCL.

I. Setting EPCIO Series Motion Control Card Hardware Parameters

The EPCIO Series motion control card hardware parameters set the control card type and the I/O address used. Please note that currently the ***MCCL does not support combining ISA Bus and PCI Bus motion control cards*.** The EPCIO Series motion control card hardware parameters are defined below:

```
typedef _SYS_CARD_CONFIG
{
    WORD        wCardType;
    WORD        wCardAddress;
    WORD        wIRQ_No;
    WORD        wPaddle;
} SYS_CARD_CONFIG;
```

wCardType: EPCIO Series motion control card type

0 Four-axis ISA Bus EPCIO Series motion control card (EPCIO-400/405)

1	Six-axis ISA Bus EPCIO Series motion control card (EPCIO-601/605)
2	Four-axis PCI Bus EPCIO Series motion control card (EPCIO-4000/4005)
3	Six-axis PCI Bus EPCIO Series motion control card (EPCIO-6000/6005)

wCardAddress: The I/O address used by the ISA Bus EPCIO Series motion control card Possible I/O addresses include: 0x200, 0x220, 0x240, 0x260, 0x280, 0x2a0, 0x2c0, 0x2e0, 0x300, 0x320, 0x340, 0x360, 0x380, 0x3a0, 0x3c0, and 0x3e0.
The ISA Bus EPCIO Series motion control card users must go to WINDOWS [Control Panel] -> [System] -> [Device Manager] -> [Inspection] to find a configurable (unoccupied) I/O address for the EPCIO Series motion control card.
The ISA Bus EPCIO Series motion control card users also need to adjust the DIP switch on the card to correspond with the I/O address. The PCI Bus EPCIO Series motion control card users can ignore this parameter setting.

wIRQ_No: Interruption number used in the EPCIO Series motion control card
ISA Bus EPCIO Series motion control card users must go to WINDOWS [Control Panel] -> [System] -> [Device Manager] -> [Inspection] to find a configurable (unoccupied) IRQ number for the EPCIO Series motion control card. The PCI Bus EPCIO Series motion control card users can ignore this parameter setting.

wPaddle: To preserve the field, set to 0.

If at this time an EPCIO-405 motion control card and an EPCIO-605 motion control card are required, the motion control card hardware parameters could be set to the following:

```
SYS_CARD_CONFIG stCardConfig[] = {{0, 0x200, 5, 0}, {1, 0x240, 7, 0}};
```

This parameter will be transmitted with the other parameters when MCC_InitSystem() is called. At this time, the EPCIO-405 motion control card index is 0, and the EPCIO-605 motion control card index is 1.

If an EPCIO-4005 motion control card and an EPCIO-6000 motion control card are used, the motion control card hardware parameters could be set to the following:

SYS_CARD_CONFIG stCardConfig[] = {{2, 0, 0, 0}, {3, 0, 0, 0}};

This parameter will be transmitted with the other parameters when MCC_InitSystem() is called. At this time, the EPCIO-4005 motion control card index is 0, and the EPCIO-6000 motion control card index is 1.

II. Enabling the MCCL

Use MCC_InitSystem() to initiate the MCCL. MCC_InitSystem() command declaration is as follows:

```
int MCC_InitSystem(     int                 nInterpolateTime ,
                        SYS_CARD_CONFIG     * psCardConfig ,
                        WORD                wCardNo );
```

nInterpolateTime is the interpolation time (please refer to the explanation in a later section) in units of ms. The setting limits are between 1 ms to 50 ms, with a suggested value of 5 ms. Shorter interpolation times will reduce the distance between two interpolation points, but will increase the work load of the CPU. Below is a reference for interpolation time settings. These suggested values are not absolute, and should be adjusted according to actual needs.

System Characteristics	Suggested Interpolation Time
Vision function and low CPU efficiency	10 ms
Only requires linear motion	5 ms - 10 ms
Generally includes curved motion	5 ms
Requires curved motion trajectories to be circular	1 ms – 3 ms

psCardConfig is the EPCIO Series motion control card hardware parameter settings for the previous step. *wCardNo* is the number of the EPCIO Series motion control cards used at this time. The following is an example when using two EPCIO-601 motion control cards simultaneously.

SYS_CARD_CONFIG stCardConfig[] = {{1, 0x200, 5, 0}, {1, 0x240, 7, 0}};

MCC_InitSystem(5, stCardConfig, 2);

2.5.2 Disabling the Motion Control Command Library

To close the MCCL, simply call MCC_CloseSystem().

2.6 Motion Control

2.6.1 Position System

The Position System includes the following functions:

I. Select between absolute or incremental position system

➔ *See Also* MCC_SetAbsolute()
MCC_SetIncrease()
MCC_GetCoordType()

II. Set units to mm or inch.

➔ *See Also* MCC_SetUnit()
MCC_GetUnit()

III. Acquire current position coordinates

➔ *See Also* MCC_GetCurPos()
MCC_GetPulsePos()

IV. Enable/disable software over-travel check function

Once MCC_SetOverTravelCheck() is used to enable this function and each interpolation point has been calculated, the MCCL will check whether the interpolation point exceeds the effective work zone for each axis. If an interpolation point is determined to have exceeded the work zone, commands will not be sent to the motion control card. MCC_GetErrorCode() can be used to check the information code (for the meaning of information codes, please refer to the EPCIO Series Motion Control Command Library Reference Manual) and confirm whether the work zone has been exceeded.

➔ *See Also* MCC_GetOverTravelCheck()
MCC_GetErrorCode()

V. Enable/disable hardware over-travel check function

For this function, please refer to the description in section **2.4.1 –** "**Machine Parameter Settings.**"

➔ *See Also* MCC_EnableLimitSwitchCheck ()
MCC_DisableLimitSwitchCheck ()
MCC_GetLimitSwitchStatus()

VI. Set current system position

Users can reset the current position coordinates. After successfully calling this command, the system coordinates will move the platform to the new position.

➔ *See Also* MCC_DefinePos()

2.6.2 Basic Trajectory Planning

The MCCL provides linear, curved, circular, and helix motions (collectively referred to as general motion), in addition to point-to-point motion trajectory planning functions. Prior to using these functions, the acceleration/deceleration type (S or trapezoid), acceleration/deceleration speed, and feed speed all need to be set to machine characteristics and special requirements.

I. General Motion (linear, curved, circular, and helix motion)

General motion includes linear, curved, circular, and helical simultaneous multi-axis motion. The return values for these functions are often checked when the general motion function is used. If the return value is less than 0, the motion command is rejected. For the possible reasons for which the motion can be rejected, please refer to manuals related to return value definitions (please see "**EPCIO Series Motion Control Command Library Reference Manual**"). If the return value is greater than or equal to 0, the value is the index number given by the MCCL to the motion command. From these motion command index numbers, the user can follow the motion command execution process. Using MCC_ResetCommandIndex() resets the index value, beginning the count at zero.

A. Linear Motion

When using the linear motion command, only the destination position or displacement of each axis needs to be set. Based on the feed speed motion provided, the preset acceleration/deceleration time is 300 ms.

➔ *See Also* MCC_Line()

B. Curved Motion

When calling the curved motion command, only the reference and destination point coordinates need to be set. Based on the feed speed motion provided, the preset acceleration/deceleration time is 300 ms. The MCCL can also provide a 3-D curved motion command.

➔ *See Also*	MCC_ArcXYZ()	MCC_Arc XYZUVW()
	MCC_ArcXY()	MCC_ArcXYUVW()
	MCC_ArcYZ()	MCC_ArcYZUVW()
	MCC_ArcZX()	MCC_ArcZXUVW()

C. Circular Motion

When calling the circular motion command, only the center coordinates need to be set, and the direction of motion (clockwise or counter-clockwise) needs to be indicated. Based on the feed speed motion provided, the preset acceleration/deceleration time is 300 ms.

➔ *See Also*	MCC_CircleXY()	MCC_CircleXYUVW()
	MCC_CircleYZ()	MCC_CircleYZUVW()
	MCC_CircleZX()	MCC_CircleZXUVW()

D. Helical Motion

When calling the helical motion command, only the center coordinates and the linear feed axis destination coordinates and pitch need to be set, and the direction of motion (clockwise or counter-clockwise) needs to be indicated. Based on the feed speed motion provided, the preset acceleration/deceleration time is 300 ms.

➔ *See Also* MCC_HelicaXY_Z()

MCC_HelicaYZ_X()
MCC_HelicaZX_Y()

E. General Motion Acceleration/Deceleration Time and Feed Speed

MCC_SetAccTime() and MCC_SetDecTime() can be used to set the desired general motion acceleration/deceleration times, and MCC_SetFeedSpeed() can be used to set the desired feed speed. Additionally, note that the MCCL only considers the three axes X/Y/Z when calculating the general motion feed speed; axes U/V/W simply begin and end motion simultaneously in correspondence to the previous three axes (for linear motion). However, if there is no displacement for X/Y/Z in this motion command, then the set feed speed will become the speed of the axis among axes U/V/W that has traveled the furthest, and the other two axes will simultaneously start in concert (similar to point-to-point motion behavior).

The feed speed setting cannot exceed the limit set using MCC_SetSysMaxSpeed(). If it does exceed the limit, the value set by MCC_SetSysMaxSpeed() becomes the feed speed. MCC_SetSysMaxSpeed() must be used prior to InitSystem().

➔ *See Also* MCC_GetFeedSpeed()
MCC_GetCurFeedSpeed()
MCC_GetSpeed()

II. Point-to-Point Motion

Point-to-point motion is very similar to linear motion in general motion. The only difference is that the speed in general motion is set using MCC_SetFeedSpeed() in units of mm/s or inch/s, while point-to-point motion uses the maximum safe speed ***ratio*** with the corresponding command MCC_SetPtPSpeed(). The ratio is calculated as follows:

point-to-point feed speed for each axis =

maximum safe speed for each axis × (feed speed ratio / 100)

where

maximum safe speed for each axis = (RPM / 60) × Pitch / GearRatio

Once the feed speed for each axis is obtained, the required time for each axis can be calculated. The MCCL will then use the axis requiring the longest time as the primary axis, with the other axes starting simultaneously.

Point-to-point acceleration/deceleration time still follows the settings in general motion.

➔ *See Also* MCC_PtP()
MCC_GetPtPSpeed()

III. Pulse Motion, Inch Motion and Continuous Inch Motion

A. Pulse motion: MCC_JogPulse()

This command requires specific axis movement to be indicated in pulses (maximum displacement is 2048 pulses). When using this command, motion status must be stopped (the return value for MCC_GetMotionStatus() should be GMS_STOP).

```
MCC_JogPulse(     10,              0,              0);
                  displacement (pulses)  indicated axis  group index
```

B. Inch Motion: MCC_JogSpace()

This command requires a specified axis to move by the indicated displacement (units: mm or inch) according to the indicated feed speed ratio (please see description in point-to-point motion). While using this command, the motion status should be stopped (the return value for MCC_GetMotionStatus() should be GMS_STOP). MCC_AbortMotionEx() can be used to stop this motion. The following is an example use of this command.

```
MCC_JogSpace(    1,          20,          0,          0);
                 displacement  feed speed ratio  indicated axis  group index
```

C. Continuous Inch Motion: MCC_JogConti()

This command requires the selected axis to move according to the indicated feed speed ratio (please see the description in point-to-point motion) and direction, and will only stop at the effective work zone boundary set by the user (the definition of effective work zone is in the machine parameters). While using this command, the motion status should be stopped (the return value for MCC_GetMotionStatus() should be GMS_STOP). MCC_AbortMotionEx() can be used to stop this motion. The following is an example use of this command.

```
MCC_JogConti( 1,          20 ,          0,          0);
              displacement direction  feed speed ratio  indicated axis  group index
              (1: positive, -1: negative)
```

IV. Motion Pause, Continue, and Abort

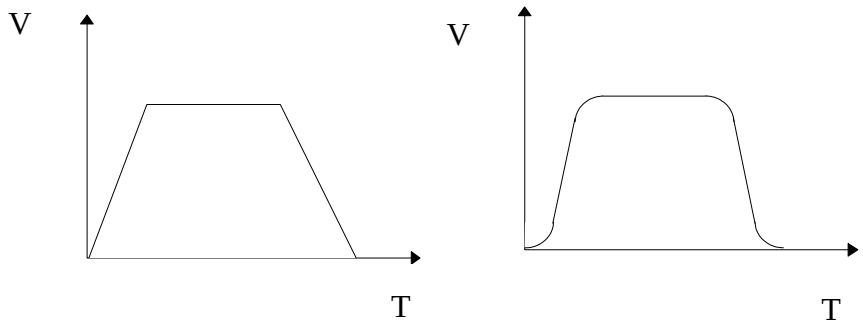
MCC_AbortMotionEx() can be used to abort all motion commands currently being executed and stored. MCC_HoldMotion() can be used to pause the motion command being executed (at which point the motion is constantly decelerated to a stop). At this time, the system will only continue executing the unfinished portion of the command after MCC_ContiMotion() is used. However, MCC_AbortMotionEx () can also be used to cancel the unfinished portions.

MCC_AbortMotionEx() stops motion using the indicated deceleration speed. If the system is already at the hold status, the deceleration time parameter is ignored.

➔ *See Also* MCC_GetMotionStatus()

2.6.3 Advanced Trajectory Planning

To achieve more elastic, efficient position control, the MCCL provides several advanced trajectory planning functions. For example, when precise positioning between different motion commands is not required and a quick arrival at the designated position is required, the motion blending function can be used. Additionally, for common tracking problems in the control system, the MCCL provides override speed, which allows dynamic adjustment to the feed speed. Below are descriptions for each of these functions.



Trapezoid Curve S Curve

Fig. 2.6.3. Acceleration/deceleration types

I. Acceleration/Deceleration Type Settings

Acceleration/deceleration type can be set as either a trapezoid curve or an S curve (see Fig. 2.6.3). The type used for each axis in point-to-point, linear, curved, circular, and helical motion is set using an identical method.

➔ *See Also*

MCC_SetAccType()	MCC_GetAccType()
MCC_SetDecType()	MCC_GetDecType()
MCC_SetPtPAccType()	MCC_GetPtPAccType()
MCC_SetPtPDecType()	MCC_GetPtPDecType()

II. Enable/Disable Motion Blending

MCC_EnableBlend() enables the motion blend function. This function can satisfy the requirements to achieve a continuous blend in speed between different motion commands (the motion does not need to decelerate and stop before the prior motion command is complete, but can directly accelerate or decelerate into the speed required for the subsequent motion command). The motion blending function includes linear-linear, linear-curved, and curved-curved motion blending.

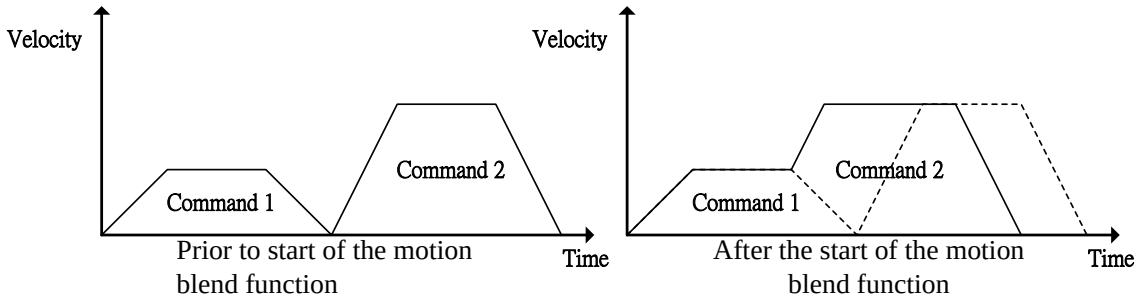


Fig. 2.6.4. Speed during motion blend

As Fig. 2.6.4 shows, when the motion blending function is enabled, the first motion command directly accelerates from its own stable speed to the stable speed of the second motion command without decelerating (the solid line in the picture on the right in Fig. 2.6.4). Using this method, the command execution time is shorter, but trajectory errors will exist at the connection points between each command. Figure 2.6.5 shows the motion trajectory for motion blending (the dotted line represents the originally planned trajectory curve).

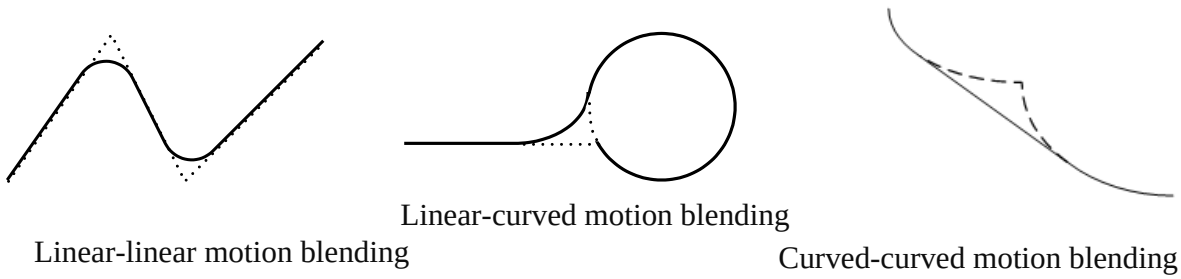


Fig. 2.6.5 Linear-linear, linear-curved, and curved-curved motion blending

➔ *See Also* MCC_DisableBlend()
MCC_CheckBlend()

III. Override Speed

Override speed can be used when the feed speed needs to be dynamically altered during motion. This function can accelerate the speed of the command being executed (V_1) to the speed required (V_2) (when $V_1 < V_2$); or decelerate from the current speed (V_3) to the required speed (V_4) (when $V_3 > V_4$).

In Fig. 2.6.6, $V_2 = V_1 \times 175 / 100$ (using MCC_OverrideSpeed(175)); similarly, $V_4 = V_3 \times 50 / 100$ (using MCC_OverrideSpeed(50)).

Using the speed ratio indicated by MCC_OverrideSpeed(), tangential speed changes will be forced. The speed ratio is defined as:

speed ratio = altered feed speed / original feed speed × 100

The original feed speed is the speed set by either MCC_SetFeedSpeed() or MCC_SetPtPSpeed(). CAUTION: ***Using MCC_OverrideSpeed() will affect all subsequent motion speeds, not only the motion being executed.***

➔ *See Also* MCC_GetOverrideRate()

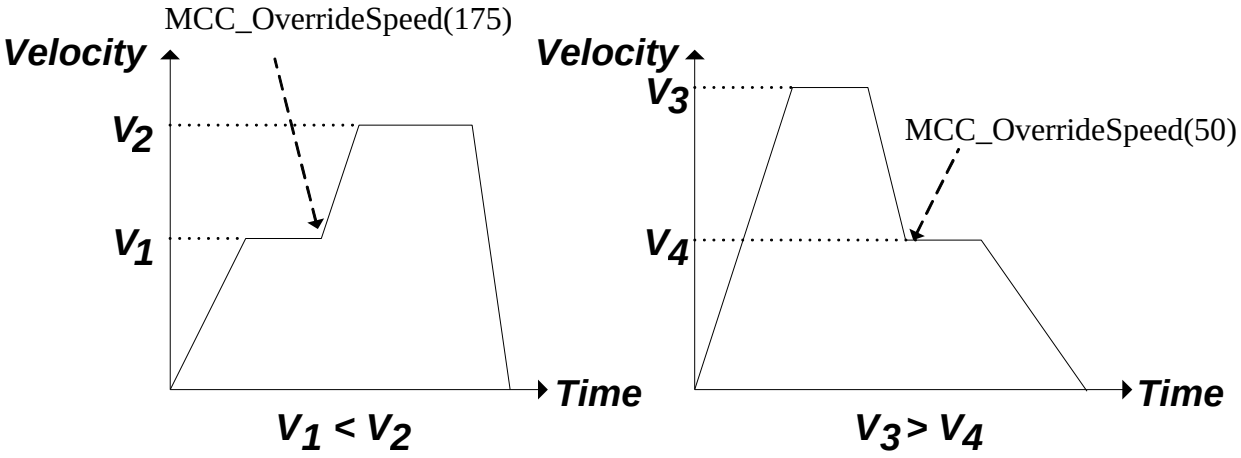


Fig. 2.6.6. Override speed

Point-to-point Override Speed:

MCC_OverridePtPSpeed() forcefully changes the speed of each axis. The parameter required for this command is the percentage of the altered speed ratio for each axis over the original speed ratio, multiplied by 100. Please see the explanation above. Using MCC_OverridePtPSpeed() will affect all subsequent motion speeds, not only the point-to-point motion being executed.

➔ *See Also* MCC_GetPtPOverrideRate()

IV. Motion Dry Run

MCC_EnableDryRun() enables the dry run function. With this function, the trajectory planning results are not sent from the motion control card, but the user can still use MCC_GetCurPos() and MCC_GetPulsePos() to acquire the content of trajectory planning. In addition to being able to obtain the motion path prior to its occurrence, the user can also utilize this information to simulate the motion trajectory on the screen.

➔ *See Also* MCC_DisableDryRun()
MCC_CheckDryRun()

V. Motion Delay

MCC_DelayMotion() forcefully delays execution of the next motion command. The delayed time is in ms; an example is displayed below:

```
MCC_Line(10, 10, 10, 0, 0, 0, 1); -------- A
MCC_DelayMotion(200, 1);
MCC_Line(15, 15, 15, 0, 0, 0, 1); -------- B
```

Once motion command A is executed, there is a 200 ms delay before continuing to execute motion command B.

➔ *See Also* MCC_GetMotionStatus()

VI. Error Message

When motion over travel occurs (the motion exceeds the software boundary), feed speed exceeds the maximum set value, acceleration/deceleration exceeds the maximum set value, curve command error occurs, or curve command execution error occurs, MCC_GetErrorCode() can obtain the error code and explain the content of the error (for error code meanings, please consult the "**EPCIO Series Motion Control Command Library Reference Manual**").

When an error occurs in a group, this group will not execute another motion command. At this point, the user must manually use MCC_GetErrorCode() to determine the reason for the error, and to remove it. MCC_ClearError() can then be used to clear the error record and return the group to normal status.

2.6.4 Interpolation Time and Acceleration/Deceleration Time

I. Setting Interpolation Time

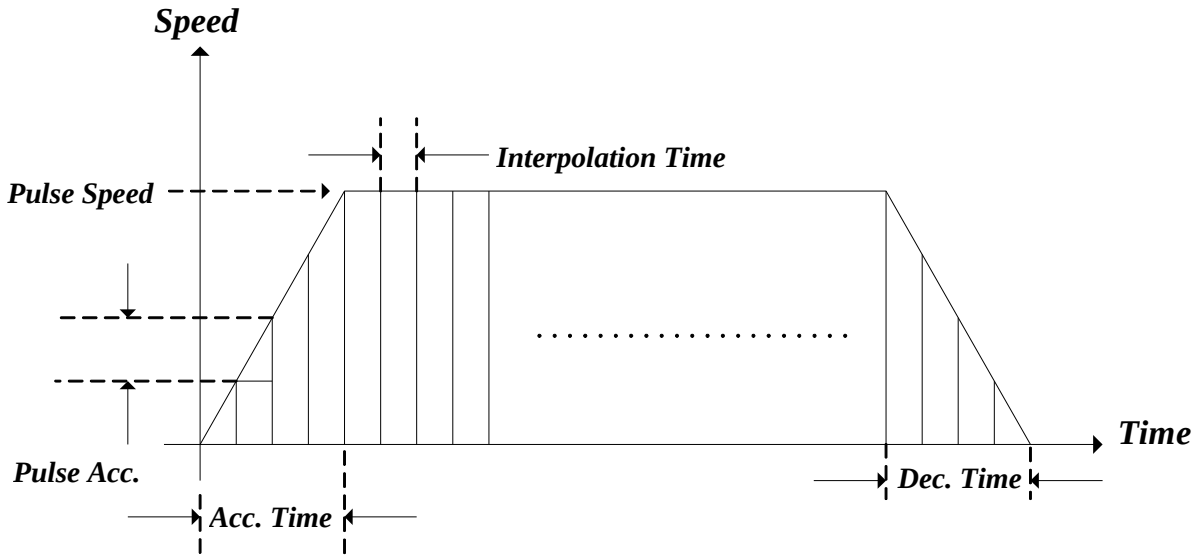


Fig. 2.6.7. Trajectory planning parameters

Interpolation time is the time gap to the next interpolation point (see Fig. 2.6.7). The minimal setting is 1 ms; the maximum setting is 50 ms.

II. Maximum Pulse Speed

The maximum pulse speed limits the number of pulses that can be sent during each interpolation time, thereby limiting the maximum feed speed. MCC_SetMaxPulseSpeed() sets the maximum pulse speed, and is set between 1 to 32767. The system default setting is 32767.

➔ *See Also* MCC_GetMaxPulseSpeed()

III. Maximum Pulse Acceleration/Deceleration

Maximum pulse acceleration/deceleration limits the maximum difference in pulses sent between neighboring interpolation times. If the acceleration/deceleration time is insufficient during the motion process, the acceleration/deceleration could

exceed machine tolerance values. This may damage the machine due to excessive motion inertia. This setting can limit the difference in pulses sent to within a range tolerable by the machine. MCC_GetErrorCode() can determine whether acceleration/deceleration has exceeded the set range during the motion process. MCC_SetMaxPulseAcc() sets the maximum pulse acceleration/deceleration between 1 to 32767. The system default setting is 32767 pulses.

➔ *See Also* MCC_GetMaxPulseAcc()

IV. Time Required for Acceleration/Deceleration

This command can either set the time needed to accelerate general or point-to-point motion to a stable speed, or set the time needed to decelerate from a stable speed to a stop. MCC_SetAccTime() and MCC_SetDecTime() set the acceleration and deceleration time needed for linear, curved, circular, and helical motion. MCC_SetPtPAccTime() and MCC_SetPtPDecTime() set the acceleration and deceleration time needed for point-to-point speed. Faster feed speeds often require longer acceleration times. Therefore, MCC_SetAccTime() and MCC_SetDecTime() are often used in combination with MCC_SetFeedSpeed(). Similarly, MCC_SetPtPAccTime() and MCC_SetPtPDecTime() are also often used in combination with MCC_SetPtPSpeed().

The below example explains the requirements for different acceleration/deceleration times for different set feed speeds. Often, users must customize the content of SetSpeed() according to machine characteristics. SetSpeed() should be used when it is necessary to change the feed speed. To avoid losing a step, MCC_SetFeedSpeed() should not be called directly, especially when using a step motor.

```
void SetSpeed(double dfSpeed)
{
    double dfAcc, dfTime;


    dfAcc = 0.04; // set acceleration to 0.04 (mm or inch)/(sec×sec)

    if (dfSpeed > 0)
    {
        dfTime = dfSpeed / dfAcc;

        MCC_SetAccTime( dfTime );
        MCC_SetDecTime( dfTime );

        MCC_SetFeedSpeed( dfSpeed);
    }
}
```

V. Acceleration/Deceleration Mode

There are two acceleration and deceleration modes: acceleration/deceleration before interpolation, and acceleration/deceleration after interpolation.

For acceleration/deceleration after interpolation, the position command first plans the displacement of each axis in terms of time, using constant speeds. It then goes through a digital filter operation to attain the position interpolation command with acceleration/deceleration. The acceleration/deceleration after interpolation mode is primarily applied to short linear segment commands. This mode can be used to solve the problem of the inability to plan the speed command motion. However, this mode will have larger in-position errors at corner trajectories, and round trajectories will be shrink. Figure 2.6.8 diagrams the acceleration/deceleration after the interpolation process.

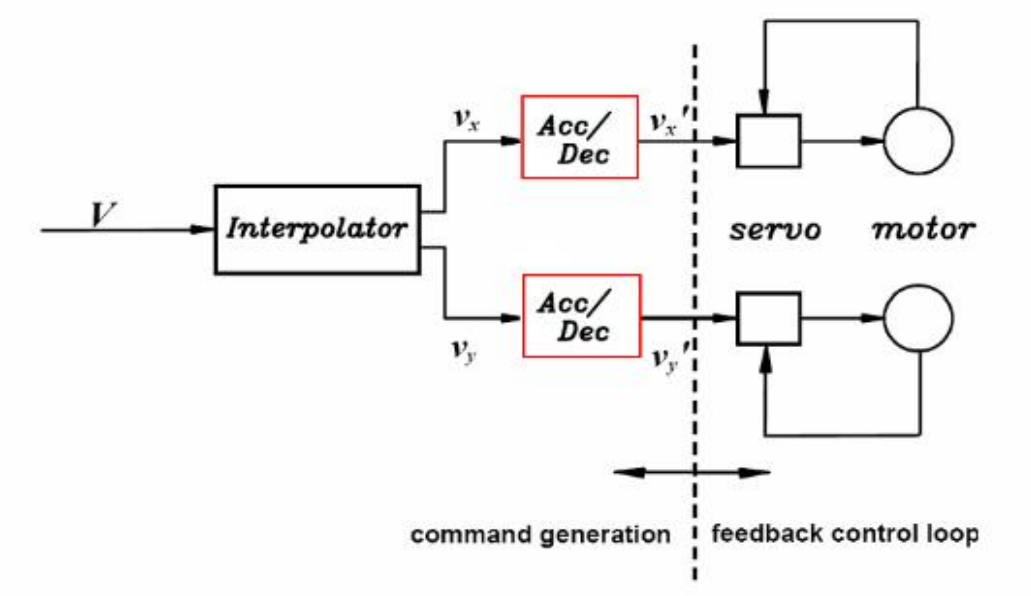


Fig. 2.6.8. Acceleration/deceleration after the interpolation process

For acceleration/deceleration before interpolation, the acceleration/deceleration parameters must first be considered during trajectory planning. First, the requirements for tangential speed acceleration/deceleration should be planned. Based on the geometric path of motion, such as linear or circular motion, the treatment sought for the speed and acceleration/deceleration of each axis follows the tangential direction of the machine's movement. Figure 2.6.9 diagrams the acceleration/deceleration before the interpolation process.

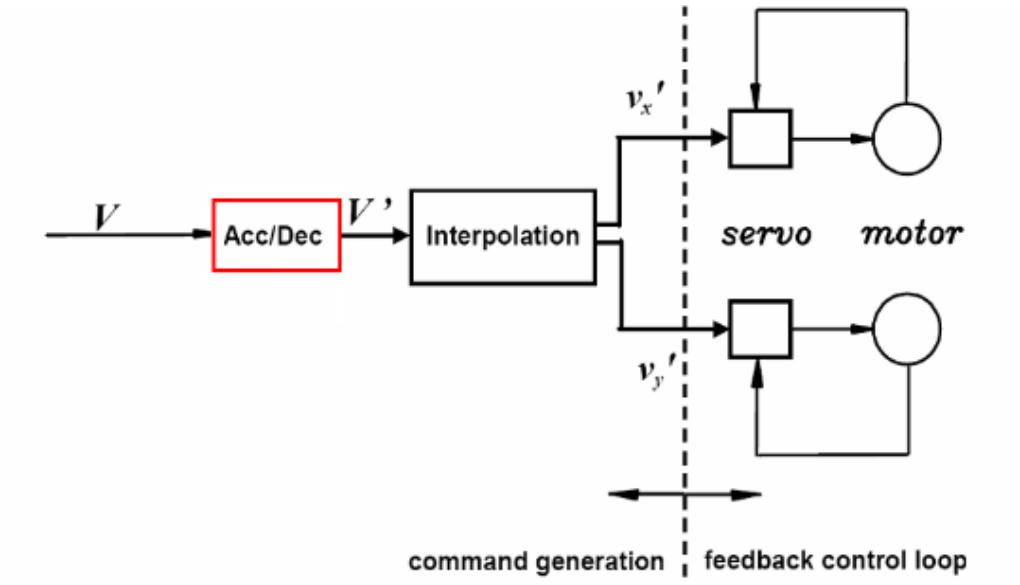


Fig. 2.6.9. Acceleration/deceleration before the interpolation process

➔ *See Also* MCC_SetAccDecMode ()

2.6.5 System Status Check

Commands provided by the MCCL can check the current actual position, estimated and actual speed, motion status, motion command stock, hardware FIFO fine motion command (FMC) stock, and the motion command being executed.

MCC_GetCurPos() obtains the current command position. Units: mm or inch

MCC_GetPulsePos() obtains the pulses sent from the control card. This value only differs from the value obtained by MCC_GetCurPos() because the latter goes through machine parameter conversion.

If an encoder is installed in the system, MCC_GetENCValue() can obtain the current actual position (the obtained value is the encoder count).

MCC_GetPtPSpeed() can obtain the feed speed ratio for point-to-point motion trajectories, while MCC_GetFeedSpeed() can obtain the feed speed for general motion. For general motion, MCC_GetCurFeedSpeed() can also obtain the current actual tangential speed, and MCC_GetSpeed() can then obtain the current actual feed speed for each axis.

The return value obtained from calling MCC_GetMotionStatus() can determine the current motion status. If the return value is GMS_RUNNING, the system is in motion. The return value GMS_STOP means that the system has stopped and there are no unexecuted stock commands. GMS_HOLD indicates that system has been paused using MCC_HoldMotion(). GMS_DELAYING means that the system is currently delayed using MCC_DelayMotion().

MCC_GetCurCommand() obtains information related to the motion commands currently being executed. The command declaration for MCC_GetCurCommand() is as follows:

MCC_GetCurCommand(*COMMAND_INFO* *pstCurCommand,
WORD wGroupIndex)

COMMAND_INFO stores content about the motion command currently being executed. It is defined as follows:

typedef struct *_COMMAND_INFO*
{
    int *nType*;
    int *nCommandIndex*;
    double *dfFeedSpeed* ;
    double *dfPos* [6];
} *COMMAND_INFO*;

where

nType: Motion command type

0. point-to-point motion
1. linear motion
2. clockwise curved, circular motion
3. counter-clockwise curved, circular motion
4. clockwise helical motion
5. counter-clockwise helical motion
6. motion delay
7. enable motion blending
8. disable motion blending
9. enable in position confirmation
10. disable in position confirmation

***nCommandIndex*:** Index for this motion command

dfFeedSpeed :

general motion	feed speed
point-to-point motion	feed speed ratio

motion delay remaining delay time (units: ms)

dfPos[]: Required destination position

MCC_GetCommandCount() obtains the motion command stock that has yet to be executed. This stock does not include the motion command currently being executed.

MCC_GetCurPulseStockCount() obtains the Fine Movement Command (FMC) stock in the EPCIO Series motion control card. During continuous motion, the FMC stock must be greater than or equal to 60 to guarantee stable motion performance. If the FMC stock is equal to 0, interpolation time must be extended (please refer to the introduction of interpolation time as described previously). Additionally, extending the interpolation time should also be considered if a lag appears in the user interface display.

2.7 In Position Control

In position control functions provided by the MCCL include:

1. Closed Loop Proportional Gain Setting
2. In Position Confirmation
3. Error Tracking Sensor
4. Handling Positional Closed Loop Control Failure
5. Gear Backlash and Gap Compensation

The following sections introduce the content and methods for each of these functions.

2.7.1 Closed Loop Proportional Gain Setting

MCC_SetPGain() sets the proportional gain parameters in the controlled closed Loop to between 1 to 16256. The method for readjusting proportional gain parameters is: after adjusting the current loop of motor drive, use the [View Profile] function in the integrated test environment (ITE) provided on the installation CD to adjust the proportional gain with error tracking (error tracking is the error between the command position and the actual position).

➔ *See Also* MCC_GetPGain()

2.7.2 In Position Confirmation

The in position confirmation function provided by the MCCL only continues to the subsequent command after confirming that the motion command being executed has arrived at its destination (within the error tolerance range). Otherwise, subsequent commands will be discarded, and an error record will be produced (which the user can choose to ignore).

To enable this function, call MCC_EnableInPos(). Once enabled, the MCCL will begin checking whether the command is in position after it sends the final FMC for the motion command. If it is in position, the next motion command will be executed. However, if the command is still not in position after waiting for the set maximum

check time (use command MCC_SetInPosMaxCheckTime() to set), subsequent commands will be discarded and an error record will be produced (for a definition of maximum check time, refer to Fig. 2.7.1).

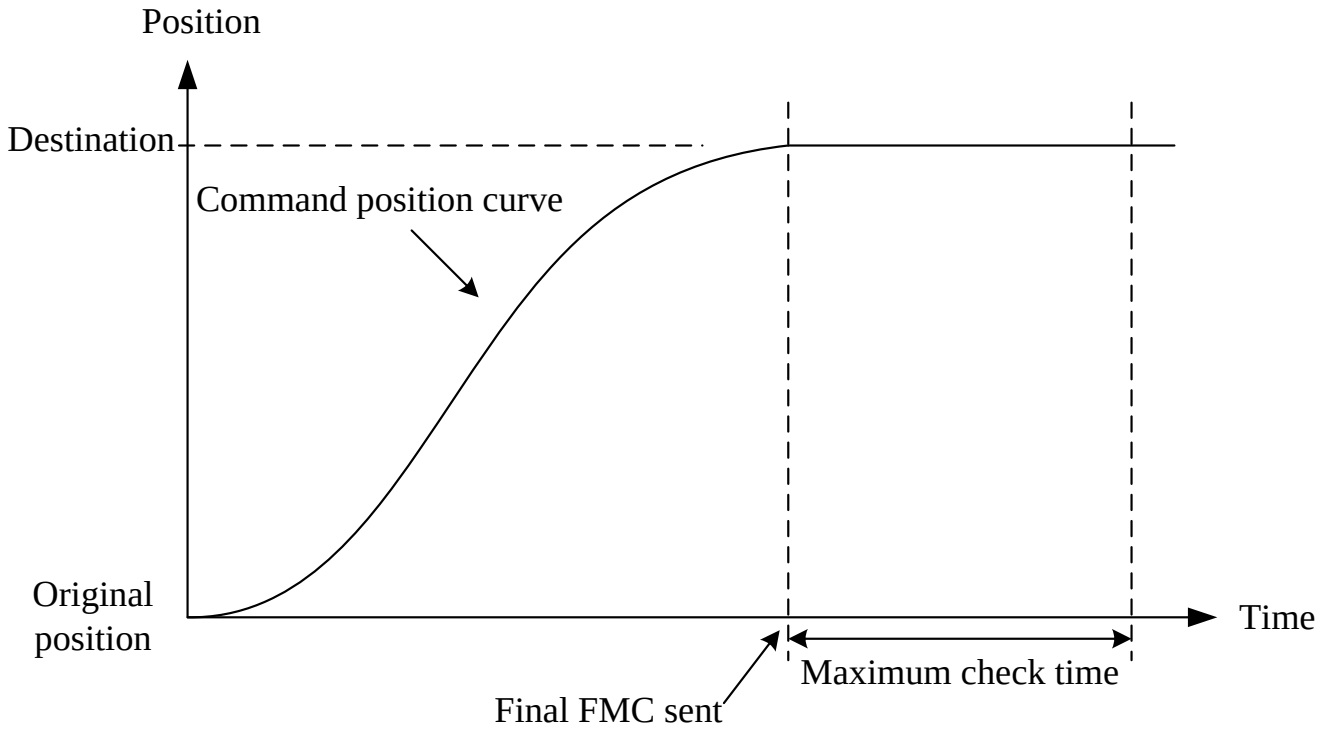


Fig. 2.7.1. Maximum check time diagram

The MCCL provides four types of the in position confirmation modes. The user can select the appropriate type using MCC_SetInPosMode(). Each mode is defined and introduced below:

Mode IPM_ONETIME_BLOCK:

When the position error for each axis in the group is less than or equal to the range of error tolerance (MCC_SetInPosToleranceEx() can be used to set this range; units: mm or inch), the in position criteria for this mode is satisfied (see Fig. 2.7.2).
If this criteria is not met prior to reaching the maximum check time, subsequent commands will be discarded, and an error record will be produced (use MCC_GetErrorCode() to obtain this record).

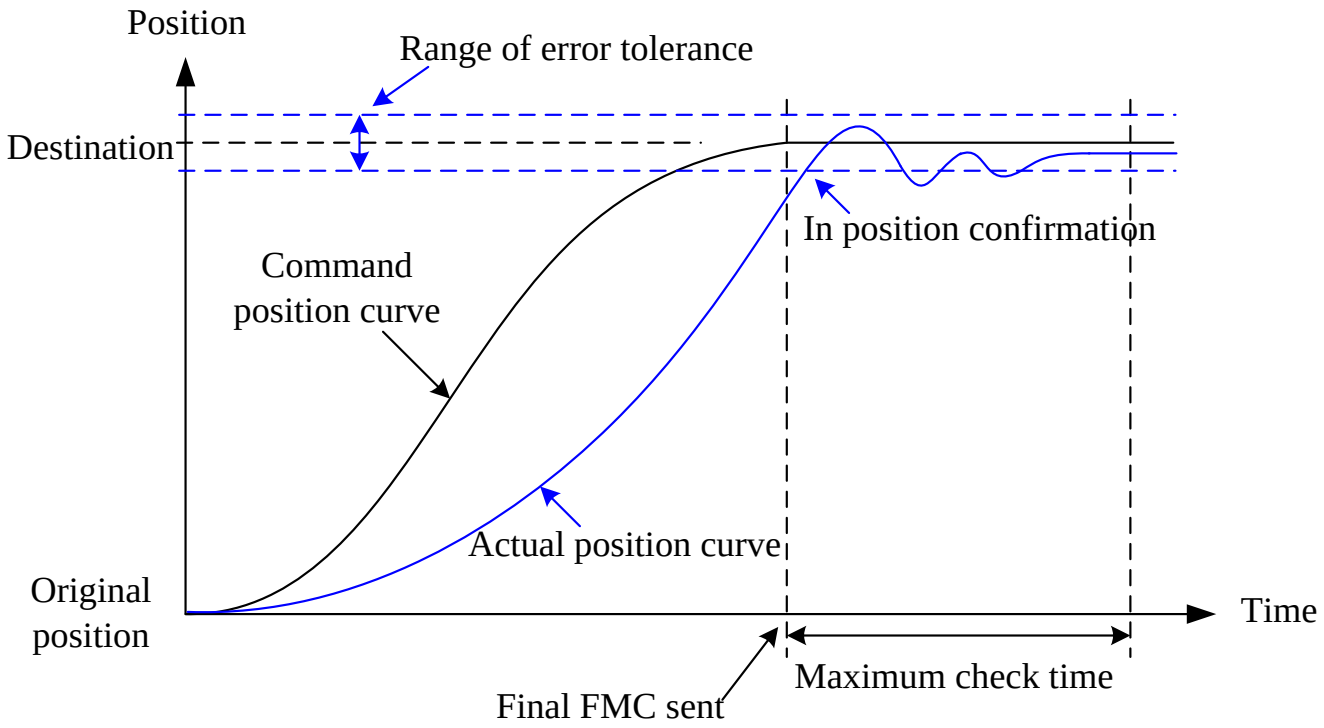


Fig. 2.7.2. IPM_ONETIME_BLOCK mode successful in position diagram

Mode IPM_ONETIME_UNBLOCK:

The in position criteria for this mode are identical to those of the IPM_ONETIME_BLOCK mode. The only difference is that if these criteria are not met prior to reaching the maximum check time, no error record is produced, and subsequent commands are directly executed.

Mode IPM_SETTLE _BLOCK:

When the position error for each axis in the group is less than or equal to the range of error tolerance (MCC_SetInPosToleranceEx() can be used to set this range; units: mm or inch) ***and remains so for a period of settle time*** (MCC_SetInPosSettleTime can be used to set this time; units: ms), the in position criteria for this mode are satisfied (see Fig. 2.7.3).

If these criteria are not met prior to reaching the maximum check time, subsequent commands will be discarded, and an error record will be produced (use MCC_GetErrorCode() to obtain this record).

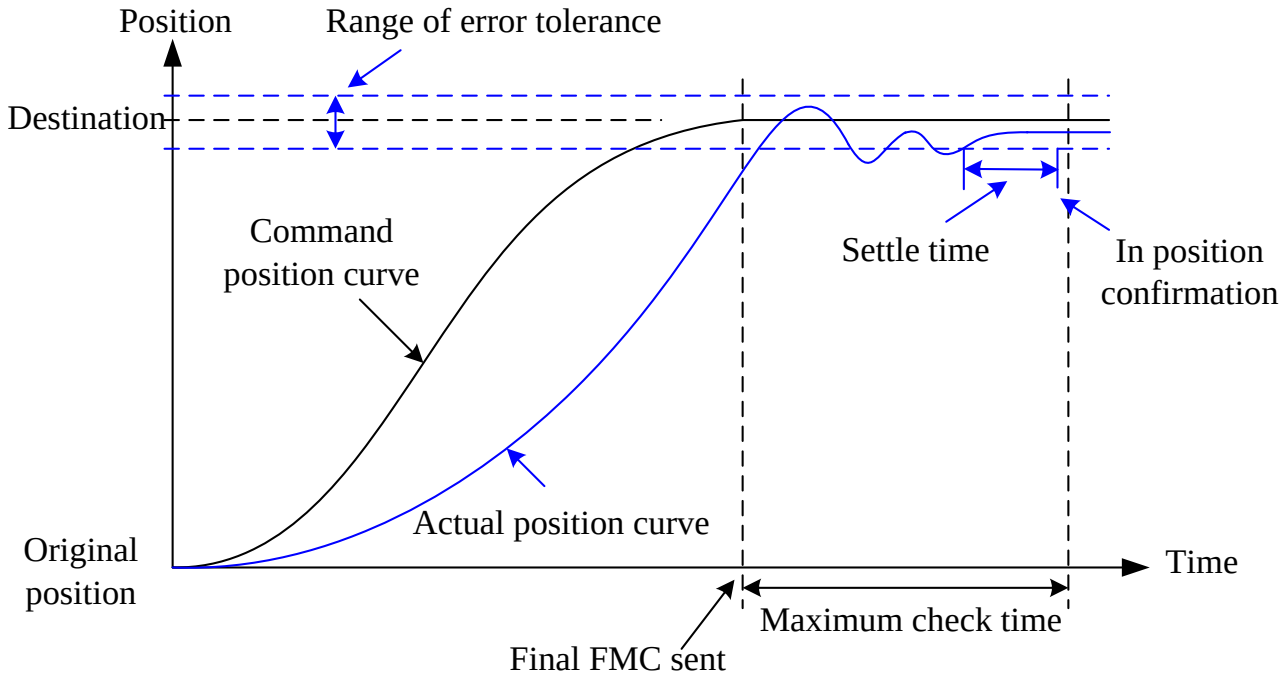


Fig. 2.7.3 IPM_SETTLE _BLOCK Mode successful in position diagram

Mode IPM_SETTLE _UNBLOCK:

The in position criteria for this mode are identical to those of the IPM_SETTLE_BLOCK mode. The only difference is that if these criteria are not met prior to reaching the maximum check time, no error record is produced, and subsequent commands are directly executed.

The greater the in position error tolerance, the shorter the time needed to complete the in position confirmation. However, the error between the motion command connection point and the trajectory path will also be greater (in the opposite situation, the error will be smaller). As Fig. 2.7.4 shows, a smaller in position error tolerance will produce a more precise trajectory (Error 1 < Error 2). Therefore, in position error tolerance should be set appropriately considering different function systems. Additionally, MCC_GetInPosStatus() can obtain the in position status of each motion axis in the group.

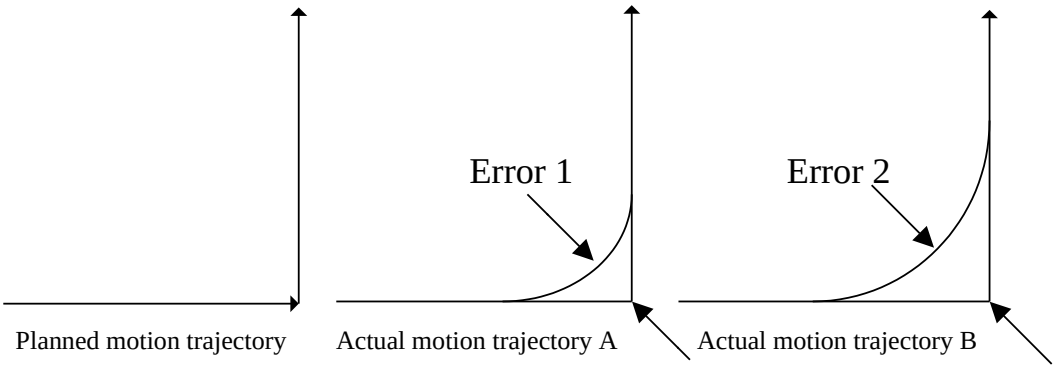


Fig. 2.7.4. Effect of the in position error on the path error

➔ *See Also* MCC_GetInPosToleranceEx()
MCC_DisableInPos()

CAUTION

1. Because the in position confirmation function compares the ***actual position*** and the target position to determine whether it is within the range of error tolerance, the motion axes that have enabled this function must also be connected to an encoder; otherwise, it will never be able to complete the in position confirmation.
2. Once the system has been determined to be successfully in position, further in position determination will not be conducted (meaning it will hold the in position status, even if the actual position leaves the set error tolerance again; see Fig. 2.7.2), until a new motion command is selected.

2.7.3 Tracking Error Sensor

The error between the motion axis command position and actual position in any given moment is referred to as the tracking error (see Fig. 2.7.5).

Under normal circumstances, the tracking error size is related to the machine characteristics, closed loop proportional gain, and motion acceleration. Excessively large error tracking means that the motion has veered (or lagged) too far from the trajectory path, and may even have had a collision.

To use this function, use MCC_SetTrackErrorLimit() to set the error tolerance range and use MCC_EnableTrackError() to enable the function. After the function has been enabled, once the tracking error for a motion axis exceeds the range, the

subsequent command trajectories for this group are stoped, and an error record is produced. The user can obtain the error code (0xF801 to 0xF806 represent excessively large tracking errors for axes X, Y, Z, U, V, W, respectively) using MCC_GetErrorCode().

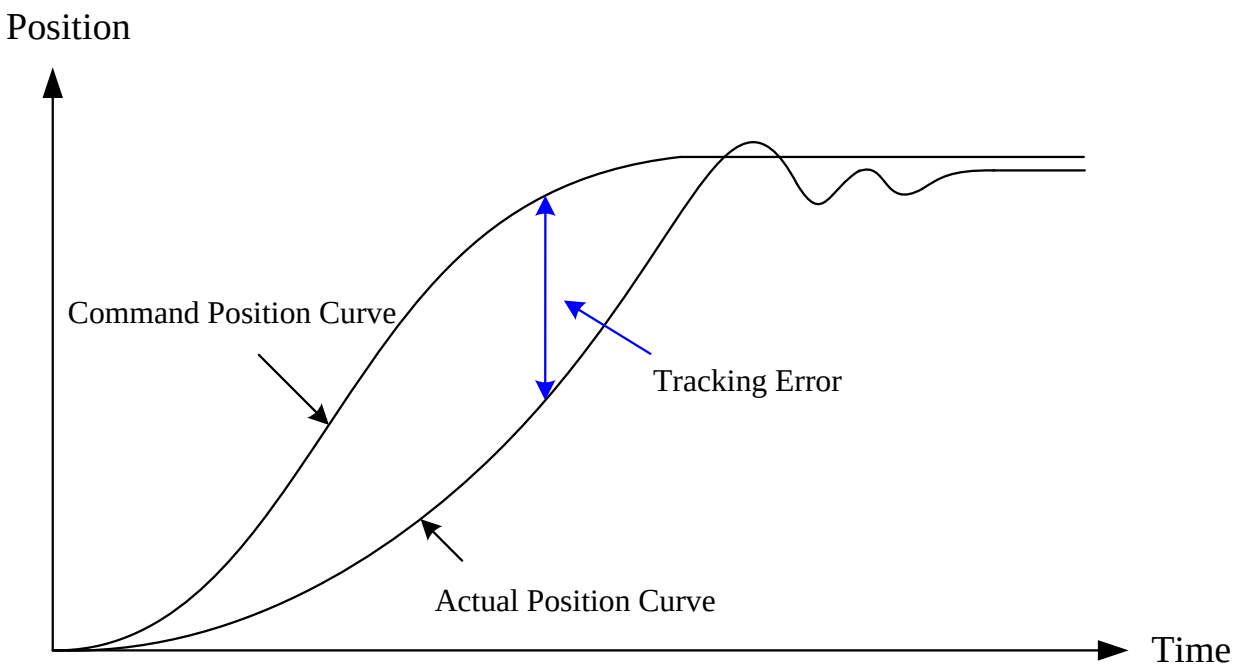


Fig. 2.7.5. Tracking error diagram

➔ *See Also* MCC_DisableTrackError()
MCC_GetTrackErrorLimit()

NOTE

When using the MCCL, any non-zero return value for MCC_GetErrorCode() indicates that the group has produced an error record. The method for handling this is as follows:

1. Determine the type of error and conduct the corresponding error removal (user should manually define this)
2. Call MCC_ClearError() to clear the error record
3. The system continues normal operations

2.7.4 Handling Positional Closed Loop Control Failure

When the position closed loop control function fails because the proportional gain parameter is set incorrectly, or due to other operational reasons, the system will be in a non-controlled state. To promptly alert the user that the system is in a non-controlled state, the motion control card will automatically produce an interruption signal. The user can customize a routine that interrupts the position control closed loop and serially connects to the system. This customized routine will be called when the motion axis position closed loop control function fails, and the user can design the handling procedure into this customized routine. The procedures to use this function are outlined below:

Step 1: **Use MCC_SetPCLRoutine() to serially connect the customized interrupt service routine**

First, the customized ISR and routine declaration must be designed following the definitions below:

typedef void(_stdcall * *PCLISR*)(*PCLINT* *)

Below is a possible customized routine design:

```
_stdcall MyPCLFunction( PCLINT *pstINTSource)
{

    // determine whether the routine was triggered due to a positional closed loop control function failure in channel 0
    if (pstINTSource-> OV0 )
    {
     // handling procedure for positional closed loop control function failure in channel 0
     }
```

```
// determine whether the routine was triggered due to a positional closed loop control function failure in channel 1
if (pstINTSource-> OV1 )
{
 // handling procedure for a positional closed loop control function failure in channel 1
 }

 // determine whether the routine was triggered due to a positional closed loop control function failure in channel 2
if (pstINTSource-> OV2 )
{
 // handling procedure for a positional closed loop control function failure in channel 2
 }

// determine whether the routine was triggered due to a positional closed loop control function failure in channel 3
if (pstINTSource-> OV3 )
{
 // handling procedure for a positional closed loop control function failure in channel 3
 }

// determine whether the routine was triggered due to a positional closed loop control function failure in channel 4
if (pstINTSource-> OV4 )
{
 // handling procedure for a positional closed loop control function failure in channel 4
 }
```

// determine whether the routine was triggered due to a positional closed loop control function failure in channel 5

if (pstINTSource-> *OV5*)

{

// handling procedure for a positional closed loop control function failure in channel 5

}

A routine such as "else if (pstINTSource-> *OV1*)" cannot be used, because pstINTSource-> *OV0* and pstINTSource-> *OV1* may not be 0 simultaneously.

Next, use MCC_SetPCLRoutine(MyPCLFunction) to serially connect the customized ISR. When the customized routine is triggered during execution, transmitting the pstINTSource parameter declared as *PCLINT* in the customized routine can determine which trigger criterion was satisfied to call the customized routine . The definition of *PCLINT* is provided below:

typedef struct *_PCL_INT*
{
BYTE *OV0*;
BYTE *OV1*;
BYTE *OV2*;
BYTE *OV3*;
BYTE *OV4*;
BYTE *OV5*;
} *PCLINT*;

If the *PCLINT* field value does not equal 0, the reasons for the customized routine call, by field value, are presented below:

OV0	Channel 0 positional closed loop control function failure
OV1	Channel 1 positional closed loop control function failure
OV2	Channel 2 positional closed loop control function failure

OV3	Channel 3 positional closed loop control function failure
OV4	Channel 4 positional closed loop control function failure
OV5	Channel 5 positional closed loop control function failure

2.7.5 Gear Backlash and Gap Compensation

When the platform controls position, deficiencies created by the gear or screw will cause position error during platform movement, such as pitch or backlash error (see Fig. 2.7.6).

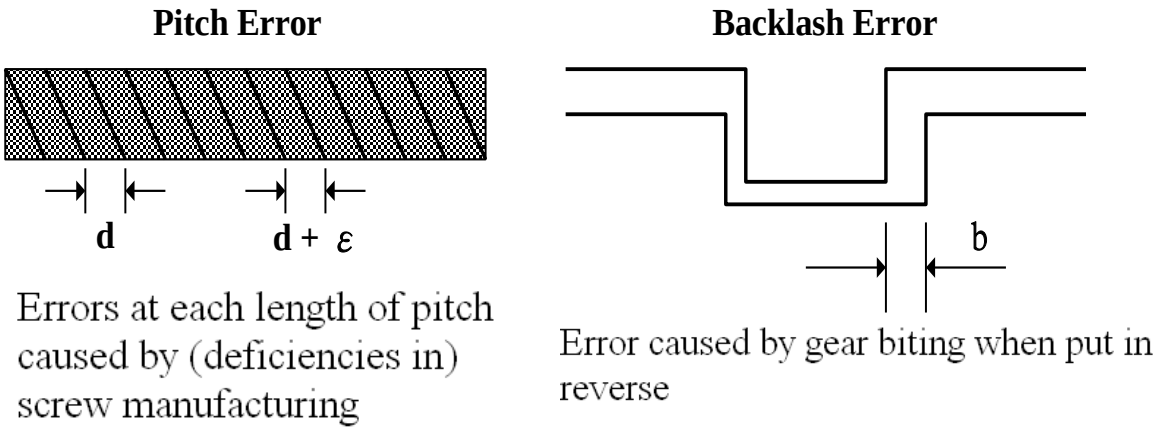


Fig. 2.7.6. Pitch and backlash error

The user can divide the platform into multiple small segments (see Fig. 2.7.7) and use a laser instrument to scan the platform back and forth once, recording the number of segment errors in a forward and backward compensation table. This compensation table is a two-dimensional array recording the amount of compensation for all compensation points in each axis. All compensation points are based on one measurement point (see Fig. 2.7.7). The user must set *dwInterval*, *wHome_No*, and the forward and backward compensation table (*nForwardTable* and *nBackwardTable*), and call the set compensation commands MCC_SetCompParam() and MCC_UpdateCompParam() to initiate the compensation function. The MCCL provides 256 compensation points for each axis. Each platform axis can be divided

into a maximum of 255 compensation segments, using the linear compensation between each segment.

When using the compensation function, compensation parameters must cover the entire platform course of work to avoid abnormal operations. Therefore, the compensation function should be enabled before completing unfinished Go Home actions. MCC_GetGoHomeStatus() can be used to whether if the Go Home actions have been completed (return value 1 indicates that the Go Home action has been completed).

To stop the compensation function, set *dwInterval* to 0. For example, execute the following program to stop Channel 0 compensation:

```
SYS_COMP_PARAM          stUserCompParam;

stUserCompParam.dwInterval = 0;

MCC_SetCompParam(&stUserCompParam, 0, 0);
MCC_UpdateCompParam();
```

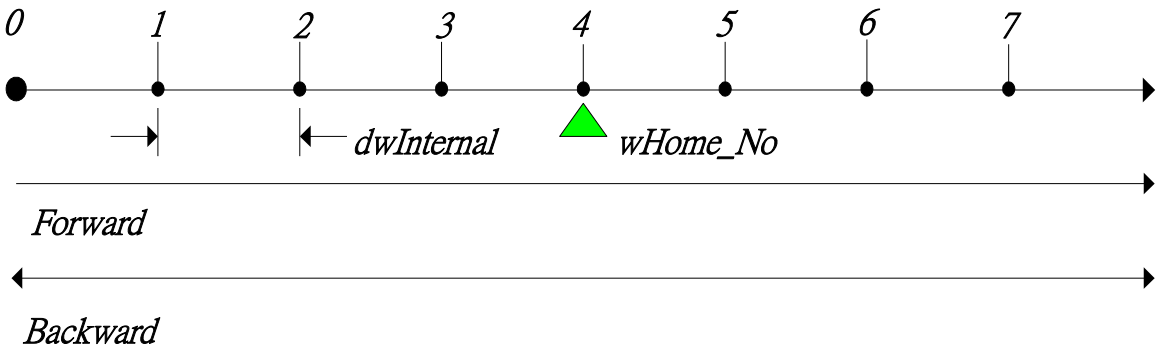


Fig. 2.7.7. Compensation segment

Compensation parameters must be set before using the compensation function. The compensation function parameters are defined below:

typedef struct *_SYS_COMP_PARAM*
{

DWORD *dwInterval*;
WORD *wHome_No*;
WORD *wPaddle*;
int *nForwardTable* [256];
int *nBackwardTable* [256];
} *SYS_COMP_PARAM*;

dwInterval:

This is the interval between compensation segments in pulses. If this value is less than or equal to 0, compensation is not performed.

wHome_No: Compensation point number for the location of each axis's home

wPaddle: Preserve field

nForwardTable: Indicator variable for forward compensation table

nBackwardTable: Indicator variable for backward compensation table

Using Fig. 2.7.7, if the X axis work area is divided into 7 compensation segments, there are a total of 8 compensation points that need to be measured (0 to 7). Home is located at compensation point 4; meaning that the system will believe it is currently at compensation point 4 after Go Home is complete. If *dwInterval* is set to 10000(pulses), the forward work range is 10000 × (7 – 4) = 30000 (pulses), and the backward work range is 10000 × (4 - 0) = 40000 (pulses). Machine parameters *dwHighLimit* and *dwLowLimit* must match these settings. The compensation parameter of each axis must be set separately. Below is an example of setting the X axis compensation parameters.

```
SYS_COMP_PARAM        stUserCompParam;

stUserCompParam.dwInterval              = 10000;

stUserCompParam.wHome_No                = 4;
stUserCompParam.nForwardTable[0]        = 22;// units: pulse
stUserCompParam.nForwardTable[1]        = 20;
stUserCompParam.nForwardTable[2]        = 15;
stUserCompParam.nForwardTable[3]        = 11;
stUserCompParam.nForwardTable[4]        = 0; // home position, set to 0
stUserCompParam.nForwardTable[5]        = 10;
stUserCompParam.nForwardTable[6]        = 12;
stUserCompParam.nForwardTable[7]        = 15;

MCC_SetCompParam(&stUserCompParam, 0, CARD_INDEX);
MCC_UpdateCompParam();
```

As explained above, the user can divide the platform into a maximum of 0~255 compensation segments, and conduct compensation in each segment using the linear compensation method. For example, if the X axis (currently located at point 4) needs to move 15000 pulses forward and to the right, the backlash error compensation table (see stUserCompParam) shows that this position is between the segments defined by nForwardTable[5] and nForwardTable[6] (because the position is between 10000 and 20000 pulses).The value for nForwardTable[5] is 10, nForwardTable[6] is 12, and nForwardTable[6] - nForwardTable[5] = 12 - 10 = 2; so the system must actually send a total of 15000 + 10 + (int)((15000 – 10000)/ 10000 × 2) = 15000 + 10 + 1 = 15011 pulses.

2.8 Go Home

Users can set the settings that accompany the Go Home parameters, including the Go Home order of each axis, acceleration/deceleration time, speed, direction, and mode. The Go Home parameter content is outlined below. For the meaning of each parameter, please refer to the explanation in section **2.4.3 – "Go Home Parameters**."

Go Home parameters (*SYS_HOME_CONFIG*):

typedef struct *_SYS_HOME_CONFIG*

{

WORD	*wMode*;
WORD	*wDirection*;
WORD	*wSensorMode*;
WORD	*wPaddle0*;
int	*nIndexCount*;
int	*nPaddle1*;
double	*dfAccTime*;
double	*dfDecTime*;
double	*dfHighSpeed* ;
double	*dfLowSpeed* ;
double	*dfOffset*;

} *SYS_HOME_CONFIG*;

2.8.1 Go Home Mode Description

The Go Home parameter *wMode* designates the Go Home motion use mode. The modes that require checking home sensor signals will first confirm the accuracy of the starting point before conducting the Go Home motion. In the following two situations, the starting point is inaccurate (assume that the initial Go Home direction of motion is to the right):

a. Go Home motion starting point is in the Home Sensor region (see Fig. 2.8.1 Case 2)

b. According to the direction of motion indicated, entering the Home Sensor region will be impossible, and will trigger a limit switch (see Fig. 2.8.1 Case 3)

If the above two abnormal starting point situations arise, the MCCL will implement the following handling procedure:

a. Move at the speed set in *dfHighSpeed* in the indicated direction until an emergency stop is executed by triggering a limit switch.

b. Move at the speed set in *dfHighSpeed* in the indicated direction into the Home Sensor region, and continue moving until exiting the Home Sensor region, at which point it will decelerate and stop.

c. Begin conducting the true Go Home action (*the action in Case 1*).

Cases 2 and 3 are possible in all of the various Go Home modes introduced in subsequent sections of this manual, just as they are in the modes requiring Home Sensor signal check; therefore, describing them further is not necessary. Only Case 1 requires an explanation.

Additionally, acceleration time *dfAccTime* represents the time used to accelerate from 0 to *dfHighSpeed* (or *dfLowSpeed*), and deceleration time *dfDecTime* represents the time used to decelerate from *dfHighSpeed* (or *dfLowSpeed*) to 0. The “emergency stop” is an immediate stop of the motion axis without deceleration.

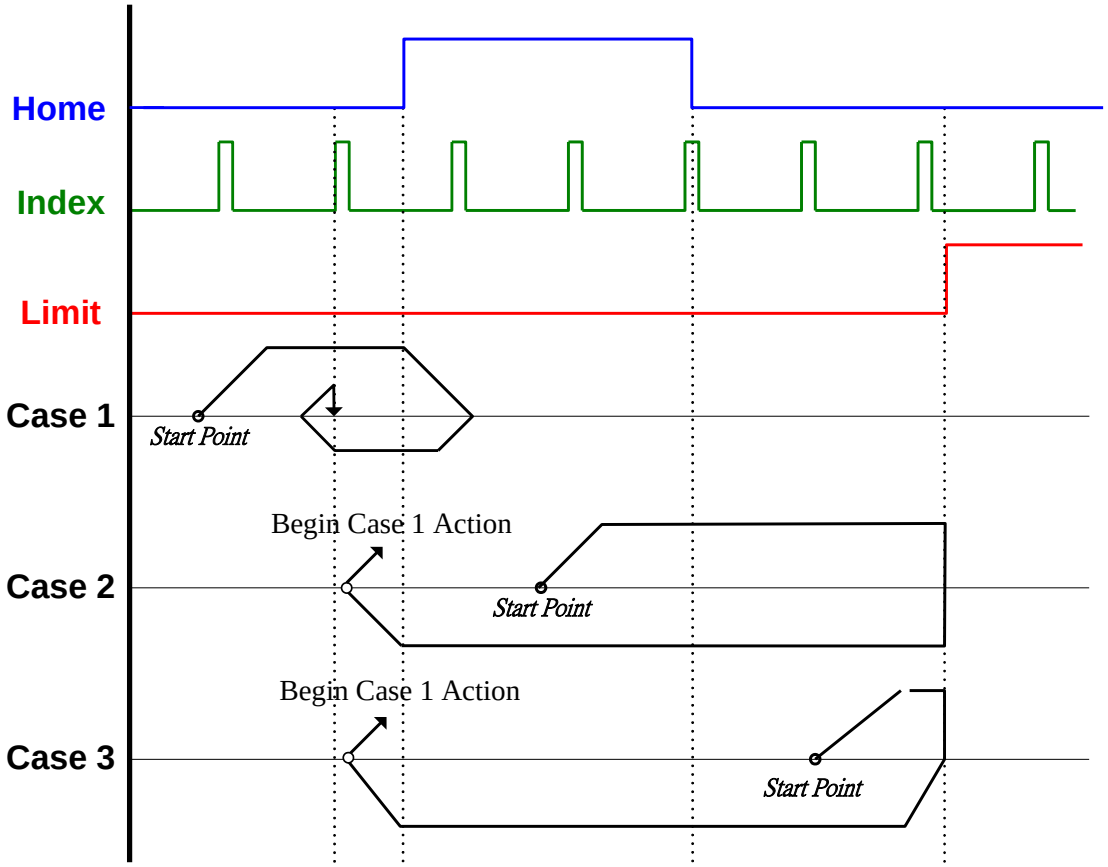
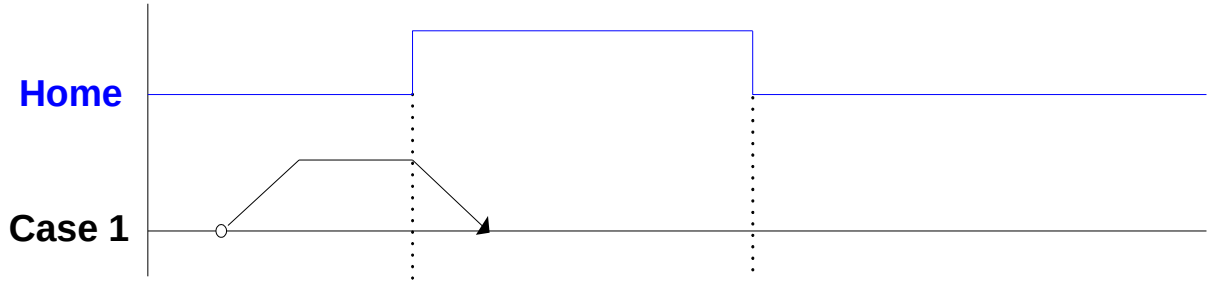


Fig. 2.8.1. Effect of different start points on Go Home

Each mode's operational characteristics are described below:

a. Mode 3 (*wMode* = 3) (Only Case 1 is described below; for cases 2 or 3, please refer to the preceding explanation)

Move at the speed set in *dfHighSpeed* in the indicated direction, decelerating to a stop upon entering the Home Sensor region, thereby completing the action (At this point, the platform will stop at the machine home, and the MCCL will move the platform to the logical home based on the parameter *dfOffset* [for details, see 2.4.1 and 2.4.3], thereby completing the entire Go Home action; all subsequent models are the same).

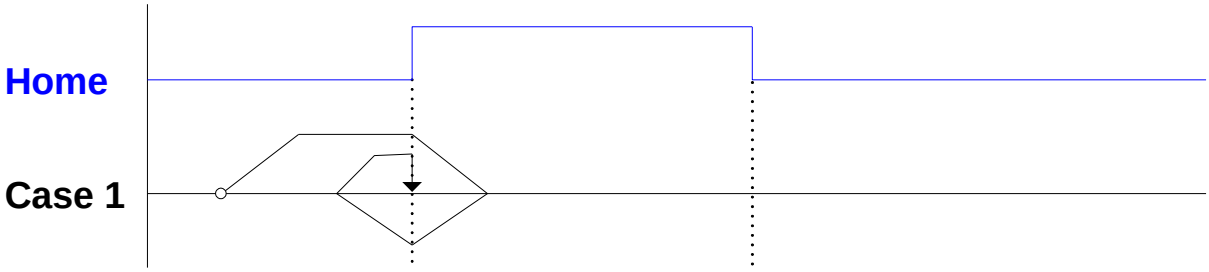


b. Mode 4 (*wMode* = 4) (Only Case 1 is described below; for cases 2 or 3, please refer to the preceding explanation)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction, decelerating to a stop upon entering the Home Sensor region.

Step 2: Move at the speed set in *dfHighSpeed* in the opposite direction, decelerating to a stop after exiting the Home Sensor region.

Step 3: Move at the speed set in *dfLowSpeed* in the indicated direction, executing an emergency stop after entering the Home Sensor region, completing the action.

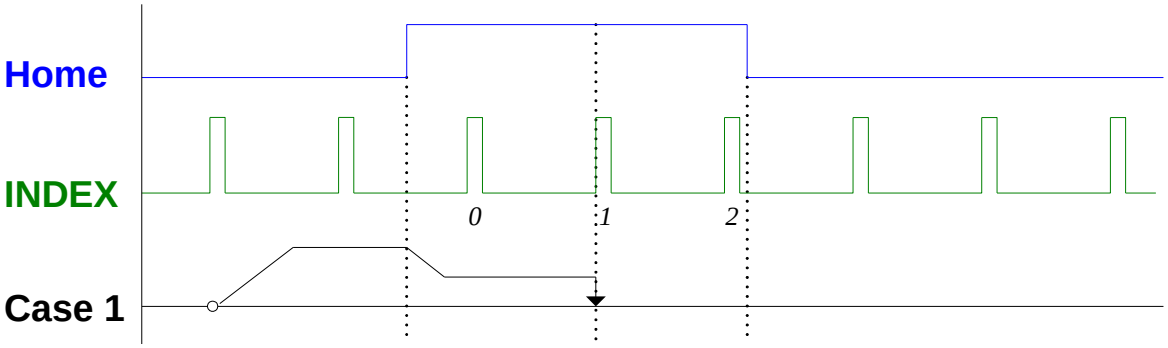


c. Mode 5 (*wMode* = 5) (Only Case 1 is described below; for cases 2 or 3, please refer to the preceding explanation)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction and begin decelerating to *dfLowSpeed* upon entering the Home Sensor region, while simultaneously searching for the

indicated index number (the example figure is set to search for index number 1, or *nIndexCount* = 1).

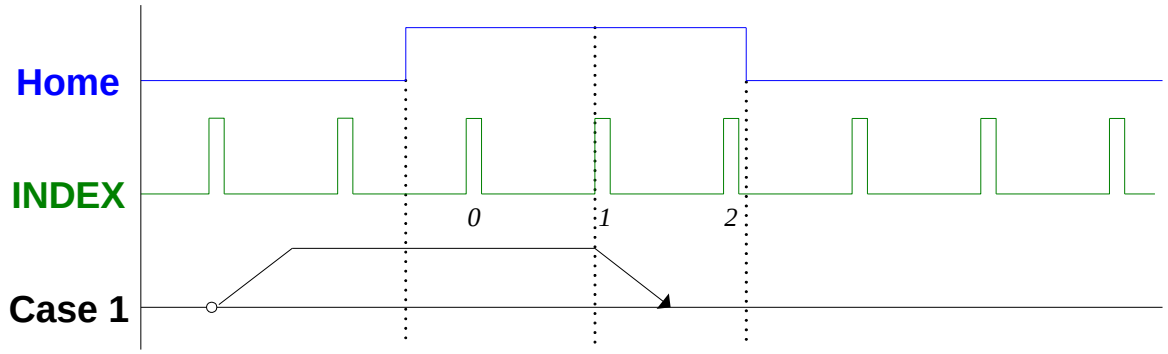
Step 2: Emergency stop after the indicated index is triggered, completing the action.



d. Mode 6 (*wMode* = 6) (Only Case 1 is described below; for cases 2 or 3, please refer to the preceding explanation)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction, searching for the indicated index number upon entering the Home Sensor region (the example figure is set to search for index number 1, or *nIndexCount* = 1).

Step 2: Decelerate to a stop after the indicated index is triggered, completing the action.

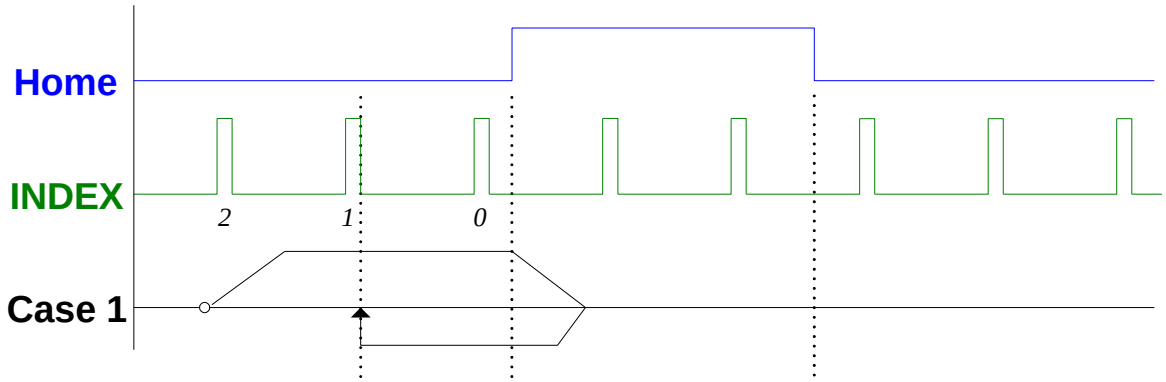


e. Mode 7 (*wMode* = 7) (Only Case 1 is described below; for cases 2 or 3, please refer to the preceding explanation)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction, decelerating to a stop upon entering the Home Sensor region.

Step 2: Move at the speed set in *dfLowSpeed* in the opposite direction, beginning to search for the indicated index number after exiting the Home Sensor region (the example figure is set to search for index number 1, or *nIndexCount* = 1).

Step 3: Emergency stop after the indicated index is triggered, completing the action.

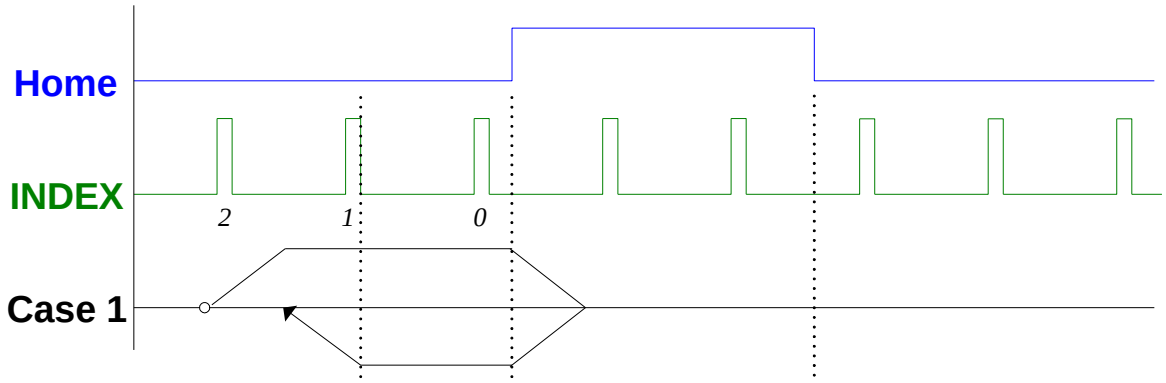


f. Mode 8 (*wMode* = 8) (Only Case 1 is described below; for cases 2 or 3, please refer to the preceding explanation)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction, decelerating to a stop upon entering the Home Sensor region.

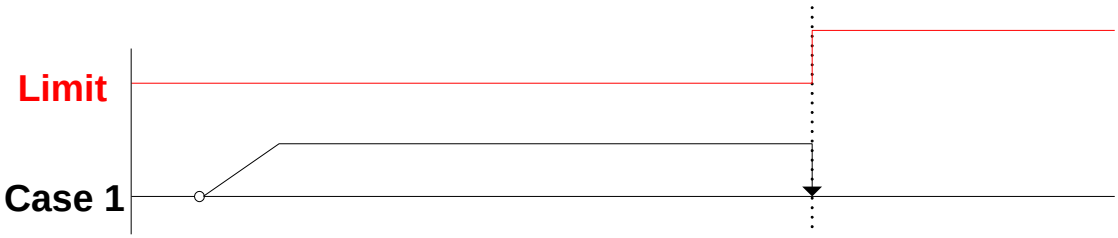
Step 2: Move at the speed set in *dfHighSpeed* in the opposite direction, beginning to search for the indicated index number after leaving the Home Sensor region (the example figure is set to search for index number 1, or *nIndexCount* = 1).

Step 3: Decelerate to a stop after the indicated index is triggered, completing the action.



g. Mode 9 (*wMode* = 9) (This mode does not have cases 2 or 3)

Move at the speed set in the *dfHighSpeed* in the indicated direction until an emergency stop is executed once a limit switch is triggered, completing the action.

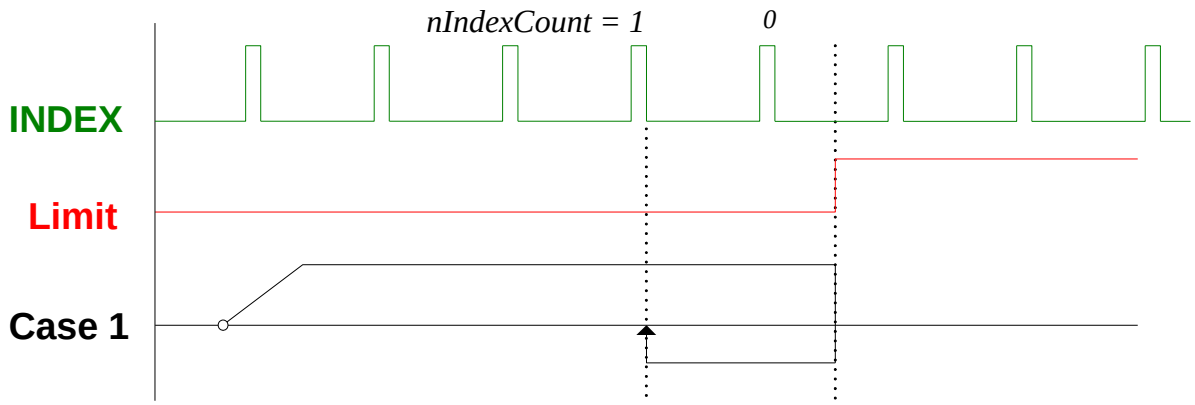


h. Mode 10 (*wMode* = 10) (This mode does not have cases 2 or 3)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction, executing an emergency stop once a limit switch is triggered.

Step 2: Move at the speed set in *dfLowSpeed* in the opposite direction, beginning to search for the indicated index number after exiting the Home Sensor region (the example figure is set to search for index number 1, or *nIndexCount* = 1).

Step 3: Emergency stop after the indicated index is triggered, completing the action.

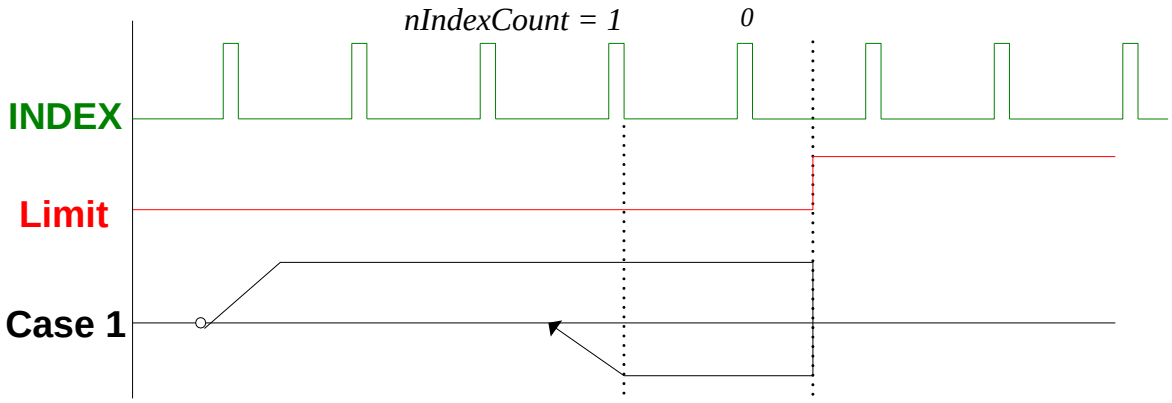


i. Mode 11 (*wMode* = 11) (This mode does not have cases 2 or 3)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction, executing an emergency stop once a limit switch is triggered.

Step 2: Move at the speed set in *dfHighSpeed* in the opposite direction, beginning to search for the indicated index number after exiting the Home Sensor region (the example figure is set to search for index number 1, or *nIndexCount* = 1).

Step 3: Decelerate to a stop after the indicated index is triggered, completing the action.

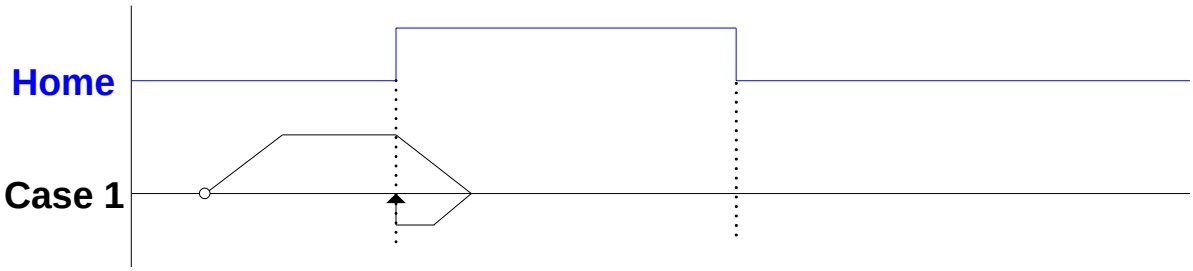


j. Mode 12 (*wMode* = 12) (Only Case 1 is described below; for cases 2 or 3, please refer to the preceding explanation)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction, decelerating to a stop upon entering the Home Sensor region.

Step 2: Move at the speed set in *dfLowSpeed* in the opposite direction to exit the Home Sensor region.

Step 3: Immediate emergency stop after leaving the Home Sensor region, completing the action.

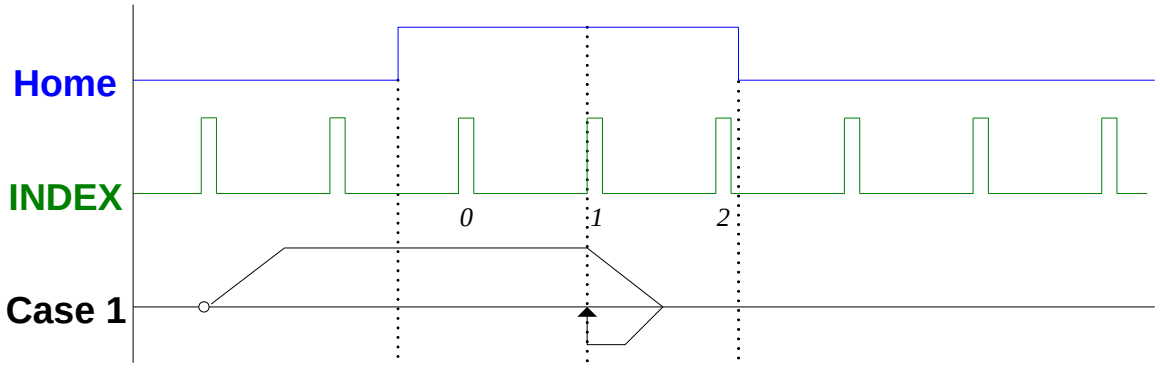


k. Mode 13 (*wMode* = 13) (Only Case 1 is described below; for cases 2 or 3, please refer to the preceding explanation)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction, beginning to search for the indicated index number upon entering the Home Sensor region (the example figure is set to search for index number 1, or *nIndexCount* = 1).

Step 2: Decelerate to a stop once the indicated index is triggered.

Step 3: Move at the speed set in *dfLowSpeed* in the opposite direction back to the position where the index was triggered, completing the action.



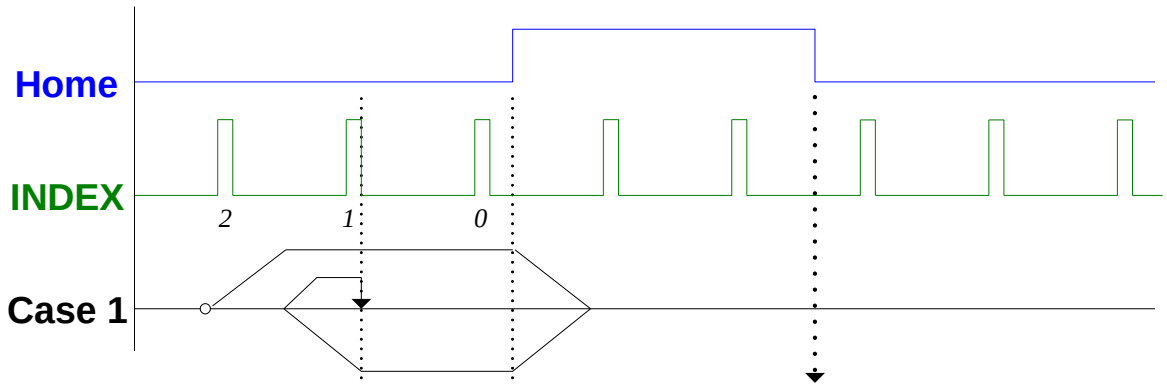
l. Mode 14 (*wMode* = 14) (Only Case 1 is described below; for cases 2 or 3, please refer to the preceding explanation)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction, decelerating to a stop upon entering the Home Sensor region.

Step 2: Move at the speed set in *dfHighSpeed* in the opposite direction, beginning to search for the indicated index number after exiting the Home Sensor region (the example figure is set to search for index number 1, or *nIndexCount* = 1).

Step 3: Decelerate to a stop once the indicated index is triggered.

Step 4: Move at the speed set in *dfLowSpeed* in the opposite direction back to the position where the index was triggered, completing the action.



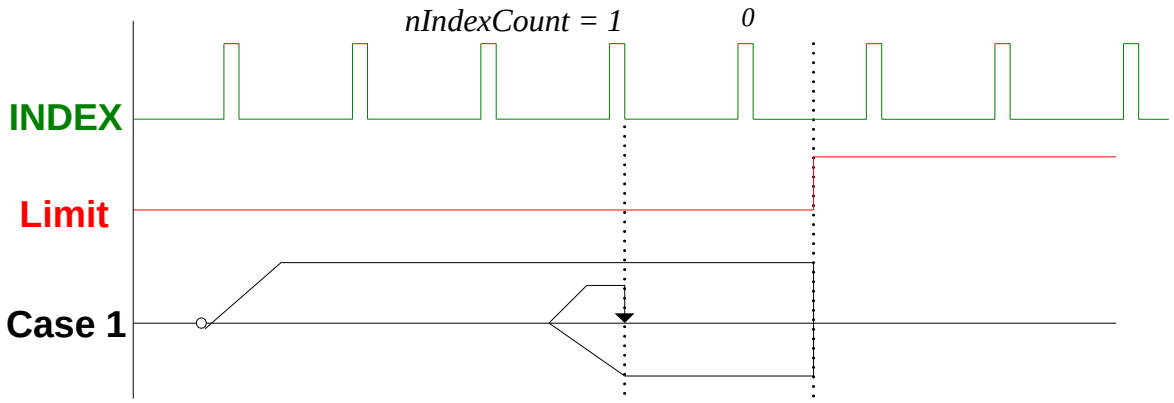
m. Mode 15 (*wMode* = 15) (This mode does not have cases 2 or 3)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction, conducting an emergency stop once a limit switch is triggered.

Step 2: Move at the speed set in *dfHighSpeed* in the opposite direction, beginning to search for the indicated index number after exiting the Home Sensor region (the example figure is set to search for index number 1, or *nIndexCount* = 1).

Step 3: Decelerate to a stop once the indicated index is triggered.

Step 4: Move at the speed set in *dfLowSpeed* in the opposite direction back to the position where the index was triggered, completing the action.

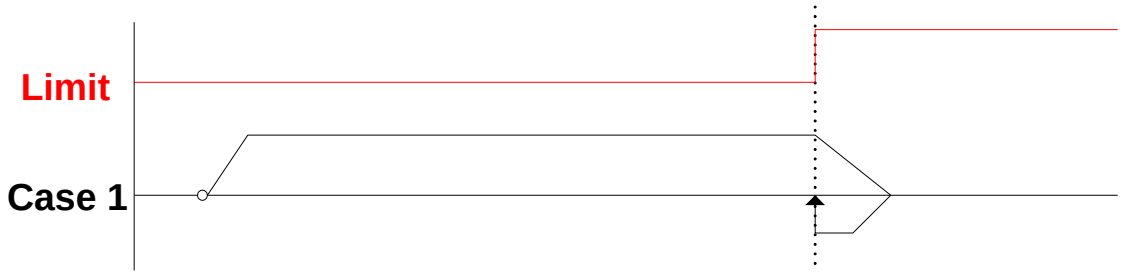


n. Mode 16 (*wMode* = 16) (This mode does not have cases 2 or 3)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction, decelerating to a stop once a limit switch is triggered.

Step 2: Move at the speed set in *dfLowSpeed* in the opposite direction to exit the limit switch region.

Step 3: Conduct emergency stop after exiting the limit switch region, completing the action.



2.8.2 Enabling Go Home

The procedure for enabling Go Home action is outlined below.

1. First, use MCC_SetHomeConfig() to set the Go Home parameters (please refer to the description in the section 2.4.3).
2. Call MCC_Home(

int *nOrder0*, int *nOrder1*, int *nOrder2*,
int *nOrder3*, int *nOrder4*, int *nOrder5*,
WORD *wCardIndex*)

where

norder0 - nOrder5	Go Home execution order for each axis
wCardIndex	motion control card index

The Go Home execution order for each axis can be set between 0 to 5, and the set value can be repeated. The MCCL will first execute the Go Home action on the motion axes with the order setting 0. Once the actions on these axes are completed, Go Home orders for motion axes set at 1 will be executed, and this principle will be followed until the Go Home actions for all motion axes are completed. An order set at 255 means that the Go Home action will not be performed for this motion axis.

MCC_AbortGoHome() can be used during the Go Home process to stop the Go Home action. The return value from MCC_GetGoHomeStatus() can also be used to know whether the Go Home action has been completed. If the value is 1, the Go Home action has been completed. If the value is 0, the Go Home action is still being performed.

CAUTION

1. Each Go Home mode can be divided into three phases:

Phase 1: Search for the Home Sensor or the limit switch

Phase 2: Search for the indicated index signal

Phase 3: Move from the machine home to the logical home

2. When multiple axes are simultaneously performing Go Home, each axis must complete Phase 1 before entering Phase 2 together. Similarly, each axis must complete Phase 2 before entering Phase 3 together. Therefore, it is possible that during the Go Home process, an axis will have completed the given phase, and must stop moving to wait for the other axes to complete the same phase. This situation is normal.

The table below lists the phases included in each Go Home mode:

Mode	Phase 1	Phase 2	Phase 3	Description
3	ˇ		ˇ	No need to execute Phase 2, but still has to wait for all axes to complete Phase 2 before proceeding together to Phase 3.
4	ˇ		ˇ	See Mode 3
5	ˇ	ˇ	ˇ	
6	ˇ	ˇ	ˇ	
7	ˇ	ˇ	ˇ	
8	ˇ	ˇ	ˇ	
9	ˇ		ˇ	See Mode 3
10	ˇ	ˇ	ˇ	
11	ˇ	ˇ	ˇ	
12	ˇ		ˇ	See Mode 3
13	ˇ	ˇ	ˇ	
14	ˇ	ˇ	ˇ	
15	ˇ	ˇ	ˇ	
16	ˇ		ˇ	See Mode 3

2.9 Local I/O Control

Local I/O refers to the input and output connections built into the EPCIO Series motion control card, and is different from the Remote I/O Module that can expand the input and output connections to 256 each (sold separately). These I/O connections have specific uses; but in practical application, if these specific uses are unnecessary (for example, if the limit switch check function or the output servo-on/off signal are not needed), these I/O connections can be used for general I/O.

2.9.1 Input Connection Status

The input connections built into the EPCIO Series motion control card include:

a. 6 (EPCIO-601/605/6000/6005) or 4 (EPCIO-400/405/4000/4005) home sensor signal input connections. Use MCC_GetHomeSensorStatus() to acquire the home sensor input signal status.

b. 6 (EPCIO-601/605/6000/6005) or 4 (EPCIO-400/405/4000/4005) positive limit switch signal input connections, and 6 (EPCIO-601/605/6000/6005) or 4 (EPCIO-400/405/4000/4005) negative limit switch signal input connections. Use MCC_GetLimitSwitchStatus() to acquire the limit switch input signal status.

c. Use MCC_GetEmgcStopStatus() to acquire the input signal status of 1 emergency stop switch signal input connection.

2.9.2 Signal Output Control

The output connections built into the EPCIO Series motion control card include:

a. 6 (EPCIO-601/605/6000/6005) or 4 (EPCIO-400/405/4000/4005) servo-on/off signal output control connections. Use MCC_SetServoOn() and MCC_SetServoOff() to output the servo-on/off signal(s).

b. 1 position ready signal control connection. Use MCC_EnablePosReady() and MCC_DisablePosReady() to output or acquire the position ready signal. Due to safety considerations, after using MCC_InitSystem() to successfully initiate the system, and after verifying that the system is operating normally, the position ready signal is also often used to enable external circuits (for example, driver or motor circuits).

2.9.3 Input Signal Triggered Interrupt Service Routine

Certain limit switch input connection signals can automatically trigger the user-customized ISR. Limit switches that can trigger the ISR include the following:

a. EPCIO-601/605/6000/6005, 7 connections:
   Channel 0 Limit Switch + (OT0+)
   Channel 1 Limit Switch + (OT1+)
   Channel 2 Limit Switch + (OT2+)
   Channel 3 Limit Switch + (OT3+)
   Channel 4 Limit Switch + (OT4+)
   Channel 5 Limit Switch + (OT5+)
   Channel 0 Limit Switch - (OT0-)

b. EPCIO-400/405/4000/4005, 7 connections:
   Channel 0 Limit Switch + (OT0+)
   Channel 1 Limit Switch + (OT1+)
   Channel 2 Limit Switch + (OT2+)
   Channel 3 Limit Switch + (OT3+)
   Channel 0 Limit Switch - (OT0-)
   Channel 1 Limit Switch - (OT1-)
   Channel 2 Limit Switch - (OT2-)

The procedure for using “input connectors triggering the ISR” is outlined below:

***Step 1*: Use MCC_SetLIORoutineEx() to serially connect the customized interrupt service routine.**

First, the customized ISR and command declaration must be designed following the definitions below:

typedef void(_stdcall **LIOISR_EX*)(*LIOINT_EX* *)

Below is a possible customized routine design:

```
stdcall MyLIOFunction( LIOINT_EX *pstINTSource)
{
    // determines whether the routine was called due to Channel 0 Limit Switch + being triggered
    if (pstINTSource-> LDI0 )
    {
            // handling procedure when Channel 0 Limit Switch + is triggered
    }

    // determines whether the routine was called due to Channel 1 Limit Switch + being triggered
    if (pstINTSource-> LDI1 )
    {
            // handling procedure when Channel 0 Limit Switch + is triggered
    }
}
```

A routine such as “else if (pstINTSource-> *LDI1*)” cannot be used, because pstINTSource-> *LDI0* and pstINTSource-> *LDI1* may not be 0 simultaneously.

Next, use MCC_SetLIORoutineEx(MyLIOFunction) to serially connect the customized ISR. When the customized command is triggered during execution, transmitting the pstINTSource parameter declared as *LIOINT_EX* in the customized

command can determine which input connection is triggered inducing calling of the customized routine. The definition of *LIOINT_EX* is provided below:

typedef struct *_LIO_INT_EX*
{
    BYTE LDI0;
    BYTE LDI1;
    BYTE LDI2;
    BYTE LDI3;
    BYTE LDI4;
    BYTE LDI5;
    BYTE LDI6;
    BYTE TIMER;
} *LIOINT_EX*;

The connections corresponding to each field in *LIOINT_EX* are defined below:

	EPCIO-601/605/6000/6005	**EPCIO-400/405/4000/4005**
LDI0	Channel 0 Limit Switch+	Channel 0 Limit Switch+
LDI1	Channel 1 Limit Switch+	Channel 1 Limit Switch+
LDI2	Channel 2 Limit Switch+	Channel 2 Limit Switch+
LDI3	Channel 3 Limit Switch+	Channel 3 Limit Switch+
LDI4	Channel 4 Limit Switch+	Channel 0 Limit Switch-
LDI5	Channel 5 Limit Switch+	Channel 1 Limit Switch-
LDI6	Channel 0 Limit Switch-	Channel 2 Limit Switch-

TIMER determines whether this command was triggered by a timer signal

If the value in one of these fields does not equal 0, then the connection corresponding to that field currently has a signal input. For example, if the MyLIOFunction() input parameter pstINTSource-> *LDI2* is not 0, it is connected to Channel 2 Limit Switch +.

Step 2: **Use MCC_SetLIOTriggerType() to set the trigger type.**

The trigger type can be set as rising edge, falling edge, or level change. Possible MCC_SetLIOTriggerType() input parameters are as follows:

LIO_INT_RISE	Rising Edge (Default)
LIO_INT_FALL	Falling Edge
LIO_INT_LEVEL	Level Change

Step 3: **Finally, use MCC_EnableLIOTrigger() to enable the "input connection signals triggering interrupt service routine" function. MCC_DisableLIOTrigger() can be used to disable this function.**

2.10 Encoder Control

The encoder control functions provided in the MCCL include feedback rate changes, count acquisition, count latch, index trigger, and automatic count comparison and trigger.

Prior to using the encoder control functions, the fields related to encoder characteristics in the machine parameters need to be accurately set. For details pertaining to these fields, please refer to the description in section **2.4.2 – "Encoder Parameters**."

2.10.1 General Control

If the encoder parameter (please see section 2.4.2) *wType* is set as ENC_TYPE_AB, meaning that the input form is set to the A/B Phase, MCC_SetENCInputRate() can be used to set the encoder feedback rate. The feedback rate can be set to 1, 2, or 4, indicating a feedback rate of ×1, ×2, or ×4, respectively. If the machine parameter wCommandMode is set to OCM_VOLTAGE (using V Command) and the feedback rate is changed, the machine value *dwPPR* needs to be reset. MCC_GetENCValue() obtains the encoder count.

2.10.2 Count Latch

The MCCL provides a "count latch" function that allows users to set the signal sources that trigger the encoder count to be recorded in the latched register. MCC_GetENCLatchValue() obtains the recorded value in the latched register. The procedure to use the "count latch" function is outlined below:

Step 1: **Use MCC_SetENCLatchSource() to set the signal source that will trigger the count latch action.**

All of the trigger sources below can trigger the encoder count to be recorded in the latched register. MCC_SetENCLatchSource() sets the trigger source criteria. Once set, multiple criteria can be obtained simultaneously. The trigger signal sources include the following:

ENC_TRIG_NO	No trigger signal source selected
ENC_TRIG_INDEX0	Index signal in encoder Channel 0
ENC_TRIG_INDEX1	Index signal in encoder Channel 1
ENC_TRIG_INDEX2	Index signal in encoder Channel 2
ENC_TRIG_INDEX3	Index signal in encoder Channel 3
ENC_TRIG_INDEX4	Index signal in encoder Channel 4 (For EPCIO-400/405/4000/4005, Manual Plus Generator (MPG) index signal)
ENC_TRIG_INDEX5	Index signal in encoder Channel 5 (EPCIO-400/405/4000/4005 does not have this signal)
ENC_TRIG_LIO0	Interrupt request from Local I/O connection OT0+
ENC_TRIG_LIO1	Interrupt request from Local I/O connection OT1+
ENC_TRIG_RDI0	Interruption in Remote I/O connection Set 0 DI 0 (Set 0 DI 0 is Digital Input 0 in Remote I/O Set 0)
ENC_TRIG_RDI1	Interrupt request from Remote I/O connection Set 0 DI1 (Set 0 DI 1 is Digital Input 0 in Remote I/O Set 0)
ENC_TRIG_ADC0	Established comparative conditions for Channel 0 ADC
ENC_TRIG_ADC1	Established comparative conditions for Channel 1 ADC

Using

MCC_SetENCTriggerSource(ENC_TRIG_INDEX0 | ENC_TRIG_LIO0, 0, 0)

means that when the encoder Channel 0 index is input and the positive direction limit for Channel 0 is triggered, the encoder count will be recorded in the latched register for Channel 0 in Card 0.

Step 2: **Use MCC_SetENCLatchType() to set the count latch mode.**

After completing ***Step 1,*** use MCC_SetENCLatchType() to set the count latch mode. The possible modes include the following:

ENC_TRIG_FIRST	The count is immediately latched and unchanged when the first trigger criterion is satisfied.
ENC_TRIG_LAST	The latch count is updated to an unlimited number when the trigger criteria are satisfied.

Step 3: **Use MCC_GetENCLatchValue() to obtain the latched register record.**

The MCCL does not have a command that can be used to determine whether the record in the latched register has been updated. However, all of the trigger sources that update the latched register record also trigger the ISR, so the user can use this function to know whether the record has been updated, and use MCC_GetENCLatchValue() to obtain the updated record. For actual application, please refer to the "**EPCIO Series Motion Control Command Library Examples Manual.**"

2.10.3 Encoder Count Triggered Interrupt Service Routine

The "encoder count triggered interrupt service routine" function provided in the MCCL sets the comparative value for encoder channels 0 to 5, and after the function has been enabled for the selected channels, automatically calls the user-customized ISR when the given channel count equals the set comparative value. The procedure for using this function is outlined below:

Step 1: **Use MCC_SetENCRoutineEx() to serially connect the customized interrupt service routine.**

First, the customized ISR and the routine declaration must be designed following the definitions below:

typedef void(_stdcall * *ENCISR_EX*)(*ENCINT_EX* *)

Below is a possible customized routine design:

```
stdcall MyENCFunction( ENCINT_EX *pstINTSource)
{

    // determine whether the routine was triggered due to the Channel 0 count equaling the comparative value
    if (pstINTSource-> COMP0 )
    {
     // handling procedure when the Channel 0 comparative value conditions are met
     }

    // determine whether the routine was triggered due to the Channel 1 count equaling the comparative value
    if (pstINTSource-> COMP1 )
    {
     // handling procedure when the Channel 1 comparative value conditions are met
    }

    // determine whether the routine was triggered due to the Channel 2 count equaling the comparative value
    if (pstINTSource-> COMP2)
    {
     // handling procedure when the Channel 2 comparative value conditions are met
    }

    // determine whether the routine was triggered due to the Channel 3 count equaling the comparative value
    if (pstINTSource-> COMP3)
    {
     // handling procedure when the Channel 3 comparative value conditions are met
    }
```

```
    // determine whether the routine was triggered due to the Channel 4 count equaling the comparative value
    if (pstINTSource-> COMP4)
    {
     // handling procedure when the Channel 4 comparative value conditions are met
    }

    // determine whether the routine was triggered due to the Channel 5 count equaling the comparative value
    if (pstINTSource-> COMP5)
    {
     // handling procedure when the Channel 5 comparative value conditions are met
    }
}
```

Language such as "else if (pstINTSource-> *COMP1*)" cannot be used, because pstINTSource-> *COMP0* and pstINTSource-> *COMP1* may not be 0 simultaneously.

Next, use MCC_ SetENCRoutineEx(MyENCFunction) to serially connect the customized ISR. When the customized routine is triggered during execution, transmitting the pstINTSource parameter declared as *ENCINT_EX* in the customized routine can determine which trigger criterion was satisfied to call the customized routine. The definition of *ENCINT_EX* is provided below:

```
typedef struct _ENC_INT_EX
{
    BYTE COMP0;
    BYTE COMP1;
    BYTE COMP2;
    BYTE COMP3;
    BYTE COMP4;
    BYTE COMP5;
    BYTE INDEX0;
```

BYTE *INDEX1*;
BYTE *INDEX2*;
BYTE *INDEX3*;
BYTE *INDEX4*;
BYTE *INDEX5*;
} *ENCINT_EX*;

If the *ENCINT_EX* field value does not equal 0, the reasons for the customized routine which is called are presented below by field value:

COMP0	Encoder Channel 0 count equals the set comparative value
COMP1	Encoder Channel 1 count equals the set comparative value
COMP2	Encoder Channel 2 count equals the set comparative value
COMP3	Encoder Channel 3 count equals the set comparative value
COMP4	Encoder Channel 4 count equals the set comparative value
COMP5	Encoder Channel 5 count equals the set comparative value
INDEX0	Triggered by encoder Channel 0 index signal
INDEX1	Triggered by encoder Channel 1 index signal
INDEX2	Triggered by encoder Channel 2 index signal
INDEX3	Triggered by encoder Channel 3 index signal
INDEX4	Triggered by encoder Channel 4 index signal
INDEX5	Triggered by encoder Channel 5 index signal

Step 2: **Use MCC_SetENCCompValue() to set the comparative value for the indicated channel number.**

Step 3: **Use MCC_EnableENCCompTrigger() to enable the function for the "encoder count triggered interrupt service routine". MCC_DisableENCCompTrigger() can be used to disable this function for the indicated channel.**

2.10.4 Encoder Index Triggered Interrupt Service Routine

The "encoder index triggered interrupt service routine" function provided in the MCCL uses the index signals for channels 0 to 5 to trigger the user-customized ISR. The procedure for using this function is outlined below:

Step 1: **Use MCC_SetENCRoutineEx() to serially connect the customized interrupt service routine.**

If MCC_SetENCRoutineEx() has not been called, please refer to the above procedure for calling the command. If MCC_SetENCRoutineEx() has been called, simply add the parameter (pstINTSource) "index signal input" field (*INDEX0~INDEX5*) determination to the customized command. Refer to the following:

```
stdcall MyENCFunction( ENCINTEX *pstINTSource)
{
    // determine whether the routine was triggered by the Channel 0 index signal
    if (pstINTSource-> INDEX0)
    {
     // handling procedure when the Channel 0 index signal is input
    }

    // determine whether the routine was triggered by the Channel 1 index signal
    if (pstINTSource-> INDEX1)
    {
     // handling procedure when the Channel 1 index signal is input
    }

    // determine whether the routine was triggered by the Channel 2 index signal
    if (pstINTSource-> INDEX2)
    {
     // handling procedure when the Channel 2 index signal is input
    }
```

```
    // determine whether the routine was triggered by the Channel 3 index signal
    if (pstINTSource-> INDEX3)
    {
     // handling procedure when the Channel 3 index signal is input
    }

    // determine whether the routine was triggered by the Channel 4 index signal
    if (pstINTSource-> INDEX4)
    {
     // handling procedure when the Channel 4 index signal is input
    }

    // determine whether the routine was triggered by the Channel 5 index signal
    if (pstINTSource-> INDEX5)
    {
     // handling procedure when the Channel 5 index signal is input
    }
}
```

Step 2: **Use MCC_EnableENCIndexTrigger() to enable the encoder index triggered interrupt service routine function for the indicated channels. MCC_DisableENCIndexTrigger() can be used to disable this function.**

This function can be combined with the “encoder count latch” function to obtain the count when the index signal is input (for a description of the “encoder count latch” function, please see section 2.10.2). MCC_GetENCIndexStatus() can also be used to determine whether the current motor position is located on the encoder index.

2.11 Digital to Analog Converter (DAC) Control

If motion axes that are required to output voltage have been programmed in the machine parameters as V Command motion axes (*nCommandMode* set as OCM_VOLTAGE), it not possible to use any of the commands related to the DAC discussed below. An incorrect returned value will be obtained after calling these commands. Particular attention should be paid to this point.

2.11.1 General Control

After initiating the MCCL with MCC_InitSystem(), MCC_SetDACOutput() can be used for the DAC, with a voltage output range between -10 V and +10 V.

Additionally, MCC_StopDACConv() can be used to stop the DAC conversion, and MCC_StartDACConv() can be used to restart the function.

2.11.2 DAC Hardware Trigger Mode

The "DAC hardware trigger mode" function provided in the MCCL can preprogram one DAC value for the selected DAC channel, and can trigger this preset voltage from a specific hardware trigger source. The procedure for using this function is outlined below:

Step 1: **Use MCC_SetDACTriggerOutput() to preprogram the DAC value.**

For example, using MCC_SetDACTriggerOutput(2.0, 1, 0) preprograms Card 0 DAC Channel 1 to output 2.0 volts.

Step 2: **Use MCC_SetDACTriggerSource() to set the hardware trigger source.**

Possible hardware trigger sources are defined below. Multiple trigger conditions can be set simultaneously. Please note that these trigger sources must come from the same motion control card.

1. DAC_TRIG_ENC0 Specified count in encoder Channel 0

2. DAC_TRIG_ENC1 Specified count in encoder Channel 1
3. DAC_TRIG_ENC2 Specified count in encoder Channel 2
4. DAC_TRIG_ENC3 Specified count in encoder Channel 3
5. DAC_TRIG_ENC4 Specified count in encoder Channel 4
6. DAC_TRIG_ENC5 Specified count in encoder Channel 5 (4-axis cards do not have this function)
7. DAC_TRIG_ADC0 ADC 0 Specified ADC 0 value
8. DAC_TRIG_ADC1 ADC 1 Specified ADC 1 value
9. DAC_TRIG_ADC2 ADC 2 Specified ADC 2 value
10. DAC_TRIG_ADC3 ADC 3 Specified ADC 3 value (4-axis cards do not have this function)
11. DAC_TRIG_ADC4 ADC 4 Specified ADC 4 value
12. DAC_TRIG_ADC5 ADC 5 Specified ADC 5 value
13. DAC_TRIG_ADC6 ADC 6 Specified ADC 6 value
14. DAC_TRIG_ADC7 ADC 7 Specified ADC 7 value (4-axis cards do not have this function)
15.DAC_TRIG_LDI0 Channel 0 Limit Switch + (OT0+) signal input
16.DAC_TRIG_LDI1 Channel 1 Limit Switch + (OT1+) signal input
17.DAC_TRIG_LDI2 Channel 2 Limit Switch + (OT2+) signal input
18.DAC_TRIG_LDI3 Channel 3 Limit Switch + (OT3+) signal input
19.DAC_TRIG_R0DI0 Remote I/O Set 0 DI0 signal input
20.DAC_TRIG_R0DI1 Remote I/O Set 0 DI1 signal input
21.DAC_TRIG_R0DI2 Remote I/O Set 0 DI2 signal input
22.DAC_TRIG_R0DI3 Remote I/O Set 0 DI3 signal input
23.DAC_TRIG_R1DI0 Remote I/O Set 1 DI0 signal input
24.DAC_TRIG_R1DI1 Remote I/O Set 1 DI1 signal input
25.DAC_TRIG_R1DI2 Remote I/O Set 1 DI2 signal input
26.DAC_TRIG_R1DI3 Remote I/O Set 1 DI3 signal input

Four-axis cards (EPCIO-400/4000) have a total of 23 hardware trigger sources. Six-axis cards (EPCIO-601/6000) have a total of 26 hardware trigger sources.

NOTE: EPCIO-405/4005/605/6005 motion control cards have no ADC or Remote I/O Set 1 functions, and therefore are unable to use the commands related to

ADC (index 7, 8, 9, 10, 11, 12, 13, and 14) and Remote I/O Set 1 (indices 23, 24, 25, and 26) described above.

ISRs related to these hardware trigger sources should also be enabled when setting the hardware trigger sources, thereby allowing the hardware trigger sources to trigger DAC. For example, using MCC_SetDACTriggerSource(*DAC_TRIG_ENC0* , 1, 2) sets the hardware trigger source for Card 2 Channel 1 DAC as the specified count for the Card 2 Channel 0 encoder. At this point, the Channel 0 "encoder count triggered ISR" function should also be enabled. In other words, MCC_SetENCCompValue() and MCC_EnableENCCompTrigger() must be used to enable the Channel 0 encoder ISR. For details on this function, please refer to section **2.10.3 – "Encoder Count Triggered Interrupt Service Routine.**" Similarly, if the hardware trigger source is set to the Limit Switch signal, MCC_SetLIOTriggerType() and MCC_EnableLIOTrigger() should also be used to enable the function triggering the ISR by input connection signals. For a description of this function, please refer to section **2.9.3 – "Input Connection Signal Triggered Interrupt Service Routine.**"

Step 3: **Use MCC_EnableDACTriggerMode() to enable this function, and MCC_DisableDACTriggerMode() to disable this function.**

2.12 Analog to Digital (ADC) Control

2.12.1 Initial Settings

The EPCIO-405/4005/605/6005 motion control cards do not have ADC functions and therefore do not support "ADC Control" functions. The following procedure must be completed before using the "ADC Control" functions:

Step 1: **Use MCC_SetADCConvType() to set the ADC converter type.**

(1) Using MCC_SetADCConvType(*ADC_TYPE_BIP*) means using a bipolar converter type, which can acquire a voltage range of -10 V to 10 V (The EPCIO-400/601) or -5 V to 5 V (The EPCIO-4000/6000).

(2) Using MCC_SetADCConvType(*ADC_TYPE_UNI*) means using a unipolar converter type, which can acquire a voltage range of 0 V to 20 V (The EPCIO-400/601) or 0 V to 10 V (The EPCIO-4000/6000).

Step 2: **Use MCC_SetADCConvMode() to set the ADC converter mode.**

(1) Using MCC_SetADCConvMode(*ADC_MODE_FREE*) means that continuous voltage acquisition is conducted. The voltage acquired will vary based on which ADC channel is used. This command must be combined with MCC_EnableADCConvChannel(). For an explanation of this function, see section 2.12.2 – "Continuous ADC Conversion."

(2) Using MCC_SetADCConvMode(*ADC_MODE_SINGLE*) indicates that only single voltage acquisition is used, unless MCC_StartADCConv() is called again; otherwise, the value acquired will not change. This command must be combined with MCC_SetADCSingleChannel(). For an explanation of this function, see section 2.12.3 – "Single Channel ADC Conversion."

2.12.2 Continuous ADC Conversion

After the initial settings above are complete, the below procedure should be followed to acquire ADC for a specific channel:

Step 1: **Call MCC_SetADCConvMode(*ADC_MODE_FREE*).**

Step 2: **Use MCC_EnableADCConvChannel() to allow the selected channel to convert ADC.**

A maximum of 8 groups of A/D channels are allowed to simultaneously convert ADC. ADC conversion only rotates in the input channels permitted. MCC_DisableADCConvChannel() prohibits ADC conversion in selected channels.

Step 3: **Use MCC_StartADCConv() to start and MCC_StopADCConv() to stop the ADC conversion function.**

Step 4: **Use MCC_GetADCInput() to acquire the ADC value.**

2.12.3 Single Channel ADC Conversion

The MCC_SetADCSingleChannel() command provided in the MCCL can select a channel to be the only channel in which ADC is converted, while all other channels stop conversion.

First, use MCC_SetADCSingleChannel() to select the channel, and call MCC_SetADCConvMode(*ADC_MODE_SINGLE*) to use the single conversion mode. After calling MCC_StartADCConv(), the selected channel will convert the ADC once. ADC will not be converted again after one conversion. The user must call MCC_StartADCConv() again to conduct another single conversion. MCC_GetADCWorkStatus() can be used during the conversion period (approximately 10 μs) to confirm completion of the conversion. Once the completion of conversion is confirmed, **MCC_GetADCInput() can be used to acquire the ADC value.**

2.12.4 Specific ADC Triggered Interrupt Service Routine

The “specific ADC triggered interrupt service routine” function provided in the MCCL sets the ADC comparative value for the selected ADC channels, and automatically calls the user-customized ISR when the function is enabled and the trigger conditions are met. The procedure for using this function is outlined below:

Step 1: **Use MCC_SetADCRoutine() to serially connect the customized interrupt service routine.**

First, the customized ISR and routine declaration must be designed following the definitions below:

typedef void(_stdcall **ADCISR*)(*ADCINT* *)

Below is a possible customized routine design:

```
_stdcall MyADCFunction(ADCINT *pstINTSource)
{
    // determine whether the routine was triggered due to the ADC value in the ADC Channel 0 satisfying the comparative criteria
    if (pstINTSource-> COMP0 )
    {
     // handling procedure when the Channel 0 comparative value conditions are met
     }

    // determine whether the routine was triggered due to the ADC value in the ADC Channel 1 satisfying the comparative criteria
    if (pstINTSource-> COMP1 )
    {
     // handling procedure when the Channel 1 comparative value conditions are met
     }
}
```

A routine such as “else if (pstINTSource-> *COMP1*)” cannot be used, because pstINTSource-> *COMP0* and pstINTSource-> *COMP1* may not be 0 simultaneously.

Next, use MCC_ SetADCRoutine(MyADCFunction) to serially connect the customized ISR. When the customized routine is triggered during execution, transmitting the pstINTSource parameter declared as *ADCINT* in the customized

routine can determine which trigger criterion was satisfied to call the customized routine. The definition of *ADCINT* is provided below:

typedef struct *_ADC_INT*
{
    BYTE *COMP0*;
    BYTE *COMP1*;
    BYTE *COMP2*;
    BYTE *COMP3*;
    BYTE *COMP4*;
    BYTE *COMP5*;
    BYTE *COMP6*;
    BYTE *COMP7*;
    BYTE *CONV*;
    BYTE *TAG*;
} *ADCINT*;

If the *ADCINT* field value does not equal 0, the reasons for the customized routine which is called are presented below by field value:

COMP0	ADC Channel 0 voltage satisfies trigger criteria
COMP1	ADC Channel 1 voltage satisfies trigger criteria
COMP2	ADC Channel 2 voltage satisfies trigger criteria
COMP3	ADC Channel 3 voltage satisfies trigger criteria
COMP4	ADC Channel 4 voltage satisfies trigger criteria
COMP5	ADC Channel 5 voltage satisfies trigger criteria
COMP6	ADC Channel 6 voltage satisfies trigger criteria
COMP7	ADC Channel 7 voltage satisfies trigger criteria
CONV	Any ADC channel completes ADC conversion
TAG	ADC tagged channel completes ADC conversion (only one channel is allowed to be tagged at any given period)

Step 2: **Consult the above explanation regarding "initial settings".**

Step 3: **Use MCC_SetADCCompValue() to set the ADC comparative value.**

Step 4: **Use MCC_SetADCCompMask() to set the ADC mask bit.**

When the voltage input is compared to the set comparative value, the smallest few bits can be masked from comparison, reducing the sensitivities of the comparator and preventing interruptions due to the ADC pulses. The parameters set by this command include the following:

ADC_MASK_NO	No ADC mask bit
ADC_MASK_BIT1	Uses 1 ADC mask bit
ADC_MASK_BIT2	Uses 2 ADC mask bits
ADC_MASK_BIT3	Uses 3 ADC mask bits

Step 5: **Use MCC_SetADCCompType() to set the ADC comparison mode.**

The ADC comparison mode sets the conditions for triggering interrupt. ADC comparison modes include the following:

ADC_COMP_RISE	The input voltage is compared from least to greatest
ADC_COMP_FALL	The input voltage is compared from greatest to least
ADC_COMP_LEVEL	The input voltage is changed and compared

Step 6: **Use MCC_EnableADCCompTrigger() to enable this function.**

Step 7: **Combine with the function "continuous ADC conversion" or "single channel ADC conversion"**

2.12.5 ADC Conversion Completion Triggered Interrupt Service Routine

There are two types of "ADC conversion completion triggered ISR" functions provided in the MCCL. Both are described below:

I. The interrupt service routine is triggered after any ADC channel completes ADC conversion. The procedures for using this function are outlined below:

Step 1: **Use MCC_SetADCRoutine() to serially connect the customized interrupt service routine.**

If MCC_SetADCRoutine() has not been called, please refer to the above procedure. If MCC_SetADCRoutine() has been called, simply add the parameter (pstINTSource) "ADC conversion completion" field (*CONV*) determination to the customized routine. Refer to the following:

```
_stdcall MyADCFunction( ADCINT *pstINTSource)
{
    // determine whether the command was triggered by the completion of ADC conversion by any ADC channel
    if (pstINTSource-> CONV )
    {
     // handling procedure when any channel completes ADC conversion
     }
}
```

Step 2: **Use MCC_EnableADCConvTrigger() to enable and MCC_DisableADCConvTrigger() to disable this function.**

II. The interrupt service routine is triggered after an ADC tagged channel completes ADC conversion. The procedures for using this function are outlined below:

Step 1: **Use MCC_SetADCRoutine() to serially connect the customized interrupt service routine.**

If MCC_SetADCRoutine() has not been called, please refer to the above procedure. If MCC_SetADCRoutine() has been called, simply add the parameter

(pstINTSource) “Tagged channel ADC conversion completion” field (*TAG*) determination to the customized routine. Refer to the following:

```
_stdcall MyADCFunction(ADCINT *pstINTSource)
{
    // determine whether the routine was triggered by the completion of ADC conversion by an ADC tagged channel
    if (pstINTSource-> TAG)
    {
     // handling procedure when the tagged channel completes ADC conversion
     }
}
```

Step 2: **Use MCC_SetADCTagChannel() to select the tagged channel.**

Step 3: **Use MCC_EnableADCTagTrigger() to enable and MCC_DisableADCTagTrigger() to disable this function.**

2.13 Timer and Watchdog Control

2.13.1 Timer Triggered Interrupt Service Routine

The length of the 24-bit timer on the EPCIO Series motion control card can be set using the MCCL. When the timer function is set and the timer ends (the value on the timer equals the set value), the user-customized ISR will be triggered, and the timer will be restarted. This process will continue until the function is disabled. The procedure for using this function is outlined below:

Step 1: **Use MCC_SetLIORoutineEx() to serially connect the customized interrupt service routine.**

If MCC_SetLIORoutineEx() has not been called, please refer to the above procedure (see section **2.9 – "Local I/O Control"**). If MCC_SetLIORoutineEx() has been called, simply add the parameter (pstINTSource) "timer ends" field (*TIMER*) determination to the customized routine. Refer to the following:

```
stdcall MyLIOFunction( LIOINTEX *pstINTSource)
{
    // determine whether the routine was triggered by the timer ending
    if (pstINTSource-> TIMER )
    {
     // handling procedure when the timer ending
     }
}
```

Step 2: **Use MCC_SetTimer() to set the timer in units of System Clock (25 ns).**

Step 3: **Use MCC_EnableTimerTrigger() to enable andMCC_DisableTimerTrigger() to disable the "timer triggered interrupt service routine" function .**

Step 4: **Use MCC_EnableTimer() to enable and MCC_DisableTimer() to disable the timer function .**

2.13.2 Watchdog Control

After the user has enabled the watchdog function, MCC_RefreshWatchDogTimer() must be used to clear the watchdog timer content before the watchdog timer ends (the watchdog timer value equals the set comparative value). Otherwise, once the watchdog time ends, the hardware will be reset. The procedure for using the watchdog function is outlined below:

Step 1: **Use MCC_SetTimer() to set the timer in units of System Clock (25 ns).**

Step 2: **MCC_SetWatchDogTimer() sets the watchdog timer comparative value.**

The watchdog timer comparative value is a 16-bit numerical value using the time in the timer as a basis. In other words, if the following programming code is used:

```
MCC_SetTimer(1000000, 0);
MCC_SetWatchDogTimer(2000, 0);
```

the Card 0 watchdog timer comparative value is set at (25 ns × 1000000) × 2000 = 50 s.

Step 3: **Use MCC_SetWatchDogResetPeriod() to set the reset signal duration.**

This command can set the duration of the watchdog generated hardware reset (24-bit numerical value, maximum value of 16777215). Units: system clock (25 ns).

Step 4: **Use MCC_EnableTimer() to enable the timer function.**

Step 5: **MCC_RefreshWatchDogTimer() must be used to clear the watchdog timer content before the watchdog timer ends.**

The user can combine this function with the “timer triggered interrupt service routine.” The user will be alerted before the watchdog resets the hardware, and will have to deal with the issue within the timer ISR.

2.14 Remote I/O Control

2.14.1 Initial Settings

Each EPCIO-405/605/4005/6005 motion control card possesses 1 Remote I/O card plug, referred to as Remote I/O Set 0. Each EPCIO-400/600/4000/6000 motion control card possesses 2 Remote I/O card plugs, referred to as Remote I/O Set 0 and Remote I/O Set 1. Remote I/O Set 0 and Remote I/O Set 1 can also be referred to as Remote I/O Master, and each can control 1 Remote I/O card (Indices EDIO-S001/2/3, collectively referred to as Remote I/O Slave). As the figure below shows (using the example of an EPCIO-6000 card), each Remote I/O card provides 64 output and 64 input connections.

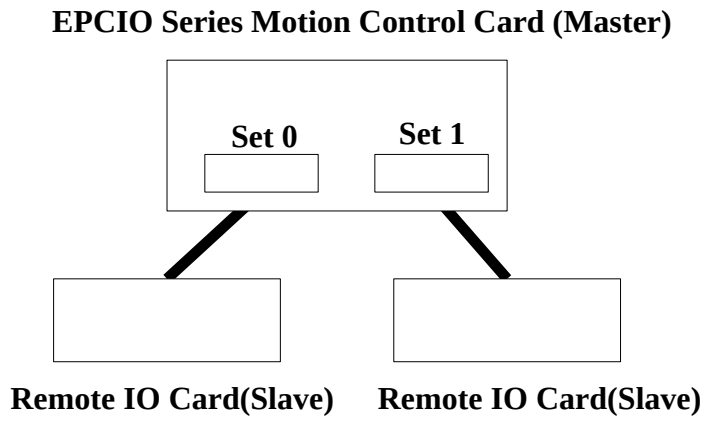


Fig. 2.14.1. Remote Master and Slave

Use EnableRIOSetControl() and EnableRIOSlaveControl() to enable the data transmission function. Below is an example where Card 1 (Index 0) Remote I/O Set 0 and its Slave data transmission function are enabled.

```
MCC_EnableRIOSetControl(RIO_SET0, 0);
MCC_EnableRIOSlaveControl(RIO_SET0, 0);
```

2.14.2 Setting and Acquiring I/O Status

When the initial settings are complete, MCC_GetRIOInputValue() can be used to acquire the input signal status, and MCC_SetRIOOutputValue() can be used to set

the output connection signal status. The prototype of MCC_GetRIOInputValue() is shown below:

```
MCC_GetRIOInputValue(   WORD*   pwValue ,
                        WORD    wSet ,
                        WORD    wPort ,
                        WORD    wCardIndex );
```

The Remote I/O acquisition function divides the 64 input connections into four ports: *RIO_PORT0*, *RIO_ PORT1*, *RIO_ PORT2*, and *RIO_ PORT3*. Each port contains 16 connections (see Fig. 2.14.2). * *pwValue* stores the statuses of the 16 input connections. Bit 0 to bit 15 in * *pwValue* store the input connections statuses for Input 0 to Input 15, respectively. Parameters *wSet* and *wPort* indicate the desired Remote I/O card set and port.

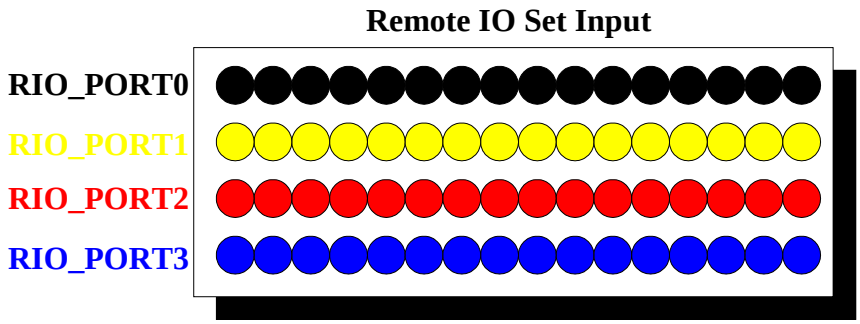


Fig. 2.14.2. 1 Remote Set contains 4 ports

The Remote I/O written input function also divides the 64 output connections into four ports: *RIO_PORT0*, *RIO_ PORT1*, *RIO_ PORT2*, and *RIO_ PORT3*. Each port contains 16 connections. Therefore, the method of use for MCC_SetRIOOutputValue() is similar to that of MCC_GetRIOInputValue(). Each time the output connections statuses are set, the statuses of the 16 output connections for the indicated port must be set simultaneously. The prototype of the command MCC_SetRIOOutputValue() is shown below:

```
MCC_SetRIOOutputValue(  WORD    wValue,
                        WORD    wSet,
                        WORD    wPort,
                        WORD    wCardIndex);
```

where wValue contains the statuses for the 16 output connections indicated by wSet and wPort.

2.14.3 Acquiring Data Transmission Status

MCC_SetRIOTransError() can be used to set the maximum number of times that Remote I/O transmission data is retransmitted. This setting is preset to the maximum value of 16. When data cannot be transmitted correctly, the EPCIO Series motion control card will retransmit the data. If the data still cannot be transmitted correctly when the set number of retransmissions is reached, the system will enter a state of data transmission error. Below is an example setting the maximum number of data retransmissions for Card 0 Set 0 to 10:

```
MCC_SetRIOTransError(10, RIO_SET0, 0);
```

MCC_GetRIOTransStatus() can be used at any time to monitor the data transmission status for each Remote I/O Set. When a data transmission error occurs, the information obtained by MCC_GetRIOMasterStatus() and MCC_GetRIOSlaveStatus() indicates the motion control card or Remote IO card from which the data transmission error originated.

2.14.4 Input Signal Triggered Interrupt Service Routine

The signals from the first four Remote I/O card input connections (*RIO_DI0*, *RIO_DI1*, *RIO_DI2*, and *RIO_DI3)* can trigger the user-customized ISR. The

procedure for using the “input connection signal triggered interrupt service routine” is outlined below:

***Step 1*: Use MCC_SetRIORoutineEx() to serially connect the customized interrupt service routine.**

First, the customized ISR and routine declaration must be designed following the definitions below:

typedef void(_stdcall * *RIOISR_EX*)(*RIOINT_EX* *)

Below is a possible customized routine design:

```
stdcall MyRIOFunction(RIOINT_EX *pstINTSource)
{
    // determine whether the routine was triggered from Set 0 Digital Input 0
    if (pstINTSource-> SET0_DI0 )
    {
        // handling procedures when Digital Input 0 signal changes
    }

    // determine whether the routine was triggered from Set 0 Digital Input 1
    if (pstINTSource-> SET0_DI1)
    {
        // handling procedures when Digital Input 1 signal changes
    }

    // determine whether the routine was triggered from Set 0 Digital Input 2
    if (pstINTSource-> SET0_DI2)
    {
        // handling procedures when Digital Input 2 signal changes
    }
```

```
    // determine whether the routine was triggered from Set 0 Digital Input 3
    if (pstINTSource-> SET0_DI3)
    {

        // handling procedures when Digital Input 3 signal changes
    }
}
```

A routine such as "else if (pstINTSource-> *SET_D1*)" cannot be used, because pstINTSource-> *SET_DI0* and pstINTSource-> *SET_DI1* may not be 0 simultaneously.

Next, use MCC_ SetRIORoutineEx(MyRIOFunction) to serially connect the customized ISR. The prototype of this routine is as follows:

```
int MCC_SetRIORoutineEx(RIOISR_EX pfnRIORoutine,
WORD wCardIndex )
```

where *pfnRIORoutine* indicates the user-customized ISR, such as MyRIOFunction; *wCardIndex* is the card index.

When the customized routine is triggered and executed, the pstINTSource parameter declared as *RIOINT_EX* transferred into the customized routine can be used to determine the input connection from which this customized routine originates. *RIOINT_EX* is defined below:

```
typedef struct _RIO_INT_EX
{
    BYTE SET0_DI0;
    BYTE SET0_DI1;
    BYTE SET0_DI2;
    BYTE SET0_DI3;
    BYTE SET0_FAIL;
    BYTE SET1_DI0;
```

BYTE *SET1_DI1*;

BYTE *SET1_DI2*;

BYTE *SET1_DI3*;

BYTE *SET1_FAIL*;

} *RIOINT_EX*;

The connections corresponding to each field in *RIOINT_EX* are defined below:

SET0_DI0	Set 0 Digital Input 0 status
SET0_DI1	Set 0 Digital Input 1 status
SET0_DI2	Set 0 Digital Input 2 status
SET0_DI3	Set 0 Digital Input 3 status
SET0_FAIL	Set 0 data transmission status
SET1_DI0	Set 1 Digital Input 0 status
SET1_DI1	Set 1 Digital Input 1 status
SET1_DI2	Set 1 Digital Input 2 status
SET1_DI3	Set 1 Digital Input 3 status
SET1_FAIL	Set 1 data transmission status

When these fields are not equal to 0, the user-customized ISR is triggered by the corresponding connection signal. For example, if the input parameter pstINTSource -> *SET0_DI0* in MyRIOFunction() is not 0, then Set 0 Digital Input 0 triggered the customized routine.

Step 2: **Set the method for Remote I/O Digital Input signals to trigger the interrupt service routine.**

MCC_SetRIOTriggerType() can be used to set the Remote I/O Digital Input trigger method as “rising edge,” “falling edge,” or “level change.” The prototype for MCC_SetRIOTriggerType() is shown below:

MCC_SetRIOTriggerType(WORD wType, WORD wSet, WORD wDigitalInput, WORD wCardIndex)

where wType is the set method for Remote I/O Digital Input signals to trigger interruption:

RIO_INT_RISE	Rising Edge Trigger
RIO_INT_FALL	Falling Edge Trigger
RIO_INT_LEVEL	Level Change Trigger

Each input connection method must be set separately, and wDigitalInput can be used to indicate the input connection desired to be set: *RIO_DI0*, *RIO_DI1*, *RIO_DI2*, and *RIO_DI3*.

Step 3: **Use MCC_EnableRIOInputTrigger() to enable the input connection signal triggered interrupt service routine function.**

Below is an example of using MCC_EnableRIOInputTrigger(), where the function is enabled for Remote I/O Set 0 Digital Input 0.

MCC_EnableRIOInputTrigger(RIO_SET0, *RIO_DI0*, 0);

2.14.5 Data Transmission Error Triggered Interrupt Service Routine

In addition to using MCC_GetRIOTransStatus(), MCC_GetRIOMasterStatus(), and MCC_GetRIOSlaveStatus() to monitor the Remote I/O data transmission status at any time, when a data transmission error occurs, it can also trigger the user-customized ISR. This function allows the user to handle data transmission errors promptly. The procedure for using this function is outlined below:

***Step 1*: Use MCC_SetRIORoutineEx() to serially connect the customized interrupt service routine.**

If MCC_SetRIORoutineEx() has not been called, please refer to the procedure described above (see the section on the "input connection signal triggered interrupt service routine"). If MCC_SetRIORoutineEx() has been called, simply add the parameter (pstINTSource) "data transmission status" field (*FAIL*) determination to the customized routine. Refer to the following:

```
stdcall MyRIOFunction( RIOINT_EX *pstINTSource)
{
    // determine whether a data transmission error has occurred
    if (pstINTSource-> SET0_FAIL )
    {
     // handling procedures when a data transmission error has occurred
     }
}
```

***Step 2*: Use MCC_EnableRIOTransTrigger() to enable the data transmission error triggered interrupt service routine function.**

The following is an example using MCC_EnableRIOInputTrigger(), where the data transmission error triggered ISR function is enabled for Remote I/O Set 0.

```
MCC_EnableRIOTransTrigger(RIO_SET0, 0);
```

2.14.6 Remote I/O Command

After calling the MCCL Remote I/O control command MCC_SetRIOOutputValue(), the input/output command will be immediately executed and interpreted for the corresponding input/output action.

After calling the MCCL I/O control command MCC_EnquRIOOutputValue(), the input/output command will not be executed immediately, but will be put in an

exclusive motion command queue. The MCCL will use the FIFO method to sequentially obtain the motion commands or input/output commands from the queue, interpreting and executing the corresponding action.

3. Editing and Translation Environment

3.1 Using Visual C++

Including Files

MCCL.h

MCCL_Fun.h

Import Library **(Users must add these files to the project)**

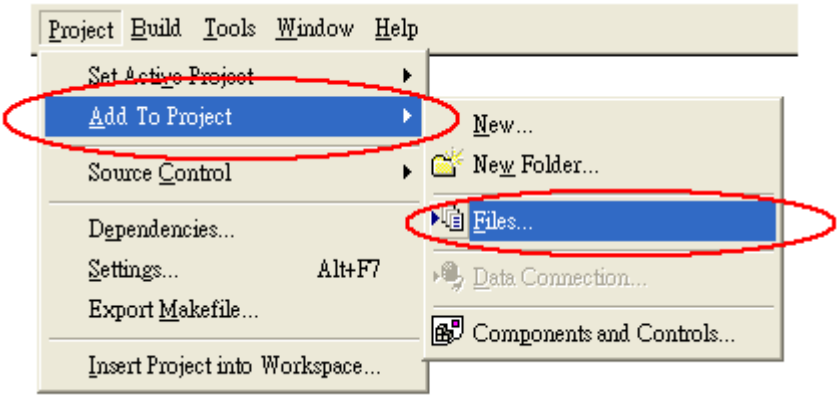
MCCLISA_50.lib (for ISA Bus)

MCCLPCI_50.lib (for PCI Bus)

Below is the process for adding the necessary import library MCCLPCI_50.lib when using an EPCIO Series PCI card (EPCIO-4000/4005/6000/6005), and using VC++ as a development tool:

Step 1:

Use **[Add To Project]** under **[Project]**



Step 2:

Select MCCLPCI_50.lib to add to project

3.2 Using Visual Basic

Including Files

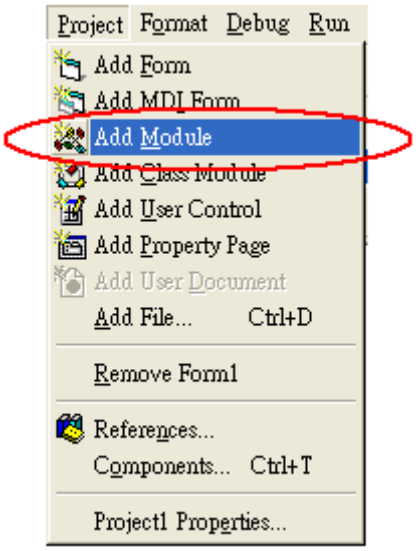
MCCLISA_50.bas (for ISA Bus)

MCCLPCI_50.bas (for PCI Bus)

Below is the process for adding the necessary import module MCCLPCI_50.bas when using an EPCIO Series PCI card (EPCIO-4000/4005/6000/6005), and using VB as a development tool:

Step 1:

Use [Add] -> [Module] under [Project]



Step 2

Select MCCLPCI_50.bas to add in the module

3.3 Using C++ Builder

Including Files

MCCL.h

MCCL_Fun.h

Import Library **(Users must add these files to the project)**

MCCLISA_50_BCB.lib	(for ISA Bus)
MCCLPCI_50_BCB.lib	(for PCI Bus)

MCCLPCI_50_BCB.lib is obtained directly using the implib command provided by BCB. The implib command form is described below:

Implib	target import library name	dll source file name

Example: implib	c:\MCCLPCI_50_BCB.lib	c:\MCCLPCI_50.dll

The MCCL dynamic link files are generated using VC++. Therefore, the import library required by BCB must use the implib command to be reproduced. The installation CD already includes a reproduced import library. The import libraries provided to BCB include:

MCCLISA_50_BCB.lib	(for ISA Bus)
MCCLPCI_50_BCB.lib	(for PCI Bus)