IMP Series
Motion Control Command Library
User Manual

Version: V.2.01

Date: 2013.01

http: //www.epcio.com.tw

Table of Contents

1. Introduction to the Motion Control Command Library

The Intelligent Motion control Platform (IMP) is equipped with built-in hard real-time operation system (VxWorks) and CPU (PowerPC 440), which use the Motion Control Command Library (MCCL). The IMP can be implemented and used under either A^+ PC or standalone modes.

A^+ PC mode refers to the mode when the users develop, compile and run applications with PC. The MCCL communicates with the IMP through PCI Bus or Ethernet. MCCL is computed by the IMP while HMI and other application commands are computed by the PC.

Applications under the A^+ PC mode can be used on platforms such as WINDOWS 98se / NT / 2000 / XP / 7 / 10 and support development environments like Visual C++, Visual Basic, and Borland C++ Builder.

For applications under the standalone mode, users can install the installation disc for the IMP to use intelligent motion control card IDK (Integrated Development Kits) for developing applications of standalone mode. For detailed standalone application integrated development environment usage, please refer to the IMP series standalone mode user manual.

The MCCL provides 3D spatial trajectory planning commands such as point-to-point, linear, arc, circular and helical motions. The MCCL also provides 14 types of operation commands: go home, dry run, delay, jog motion, pause, continue and abort motions.

For the trajectory planning function, it is possible to set different feed speeds, max feed speed, and acceleration and deceleration durations and curve types. The MCCL also consists of functions such as software and hardware over-travel check, blending, override speed, and error message handling.

For position control, users can use the MCCL to set in position loop proportional gain and error tolerance. The MCCL also provides functions like gear backlash, gear gap compensation, and in position confirmation.

For I/O signal handling, the user can utilize the MCCL to acquire home position, machine limits, and output Servo-On/Off signals. Meanwhile, certain I/O input signals can automatically trigger the Interrupt Service Routine (ISR) which can be customized by the user.

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

For D/A conversion, the user can use the MCCL to output the required voltage (-10V to 10V) as well as program the desired voltage in advance; this voltage will be output automatically when trigger conditions are met.

For A/D conversion, the user can use the MCCL to acquire the voltage input (-5V to 5V or 0V to 10V) and set the single or labeled channel voltage conversion. The ISR will be triggered automatically once the voltage conversion is completed or the voltage satisfies the comparative conditions.

For the timer function, the user can set the timer interrupt. Once the timer is enabled, when the timer has expired, it will automatically trigger the user-customizable ISR and reset the timer. This process will continue until it is disabled. The MCCL also provides a watchdog function.

The MCCL is a powerful command library that allows users to quickly develop and integrate motion control systems without requiring an in-depth understanding of trajectory planning, position control, or real-time multi-tasking environment.

2. MCCL Functions

2.1 Software Specifications

■ A^+ Operation System in PC Mode

✓ WINDOWS XP/XP Embedded

✓ WINDOWS 7/10

■ A^+ Development Environment of PC Mode

✓ Borland C++ Builder (BCB)

✓ Visual C++ (VC++)

✓ Visual Basic (VB)

✓ Visual C# (VC#)

■ Development Environment of Standalone Mode

✓IDK (provided with card)

✓WindRiver WorkBench (prepared by the user)

■ Required files when using the MCCL under A^+ PC mode

	File Name
VC++	MCCL.h MCCL_Fun.h MCCLPCI_IMP.lib
VB	MCCLPCI_IMP.bas
VC#	MCCL.cs

■ Required files when using the MCCL under standalone mode

	File Name
IDK	MCCL.h MCCL_Fun.h

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

2.2.1 Motion Axis Definition

The purpose of MCCL design is to provide motion functions that support three orthogonal axes (X-Y-Z) and five auxiliary axes (U,V,W,A,B). See Fig. 2.2.1, U,V,W,A, and B are the 5 auxiliary axes representing five independent axial directions.

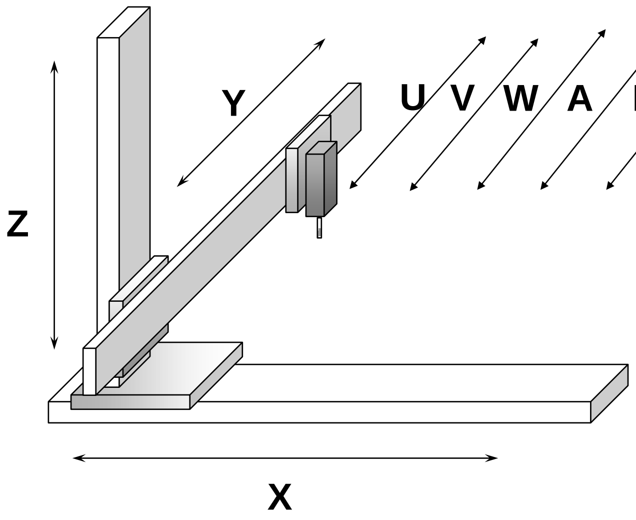


Fig. 2.2.1 Coordinate system with three orthogonal axes (**X-Y-Z**) and five auxiliary axes (**U、V、W、A、B**)

The MCCL provides a maximum of 8 control axes with synchronous motion. The users can utilize an IMP to control 1 to 8 axes simultaneously or separately. The user can set one of two coordinate modes between absolute and incremental for given motion commands. This command library will record the position relative to the home position regardless of the coordinate mode selected.

2.2.2 Maximum Number of Combinable Control Cards

Depending on the card type, each motion control card can control up to 8 groups of the system (both motor and driver). The MCCL can control up to 6 motion control cards simultaneously, thereby controlling a maximum of 48 axes. The IMP can select velocity command mode (V Command) or pulse command mode (P Command). The basic configuration is displayed in Fig. 2.2.2.

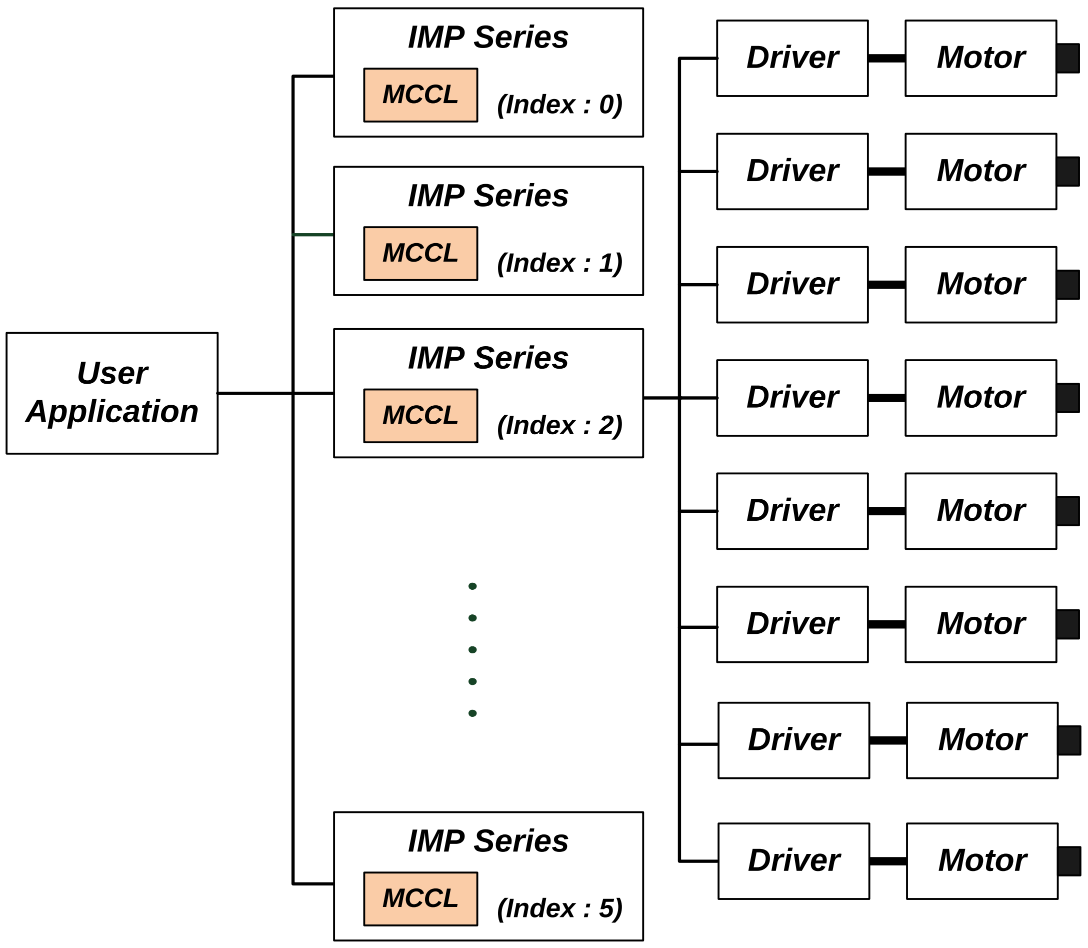


Fig. 2.2.2 The MCCL can combine with 6 IMP

2.3 Command Library Operational Properties

After motion commands in the MCCL are called, ***the related motion commands will be placed into each group's exclusive motion command queue first*** instead of being executed immediately (For descriptions regarding groups, please refer to section **2.5.1-Enabling the motion control command library**). The MCCL will then use first-in first-out (FIFO) principle to get the motion command from the queue to interpret and calculate the interpolation (see Fig. 2.3.1). However, these two operations are not sequential and synchronized, meaning that it is not necessary to wait for the completion of one motion command before the new motion command can be sent to the queue.

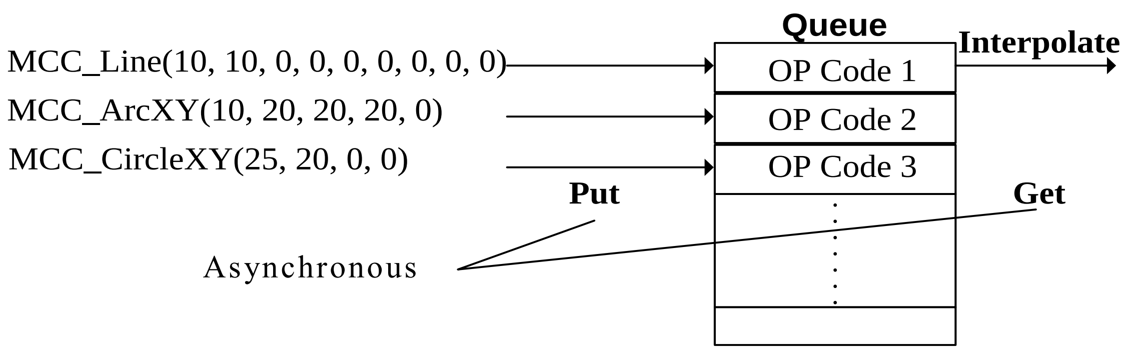

Queue
Interpolate
MCC_Line(10, 10, 0, 0, 0, 0, 0, 0, 0)
OP Code 1
MCC_ArcXY(10, 20, 20, 20, 0)
OP Code 2
MCC_CircleXY(25, 20, 0, 0)
OP Code 3
Put
Get
Asynchronous

Fig. 2.3.1 Motion command queue

The motion command queue of each group is preset to store 10,000 commands and MCC_GetCmdQueueSize can be used to acquire the current queue size. Caution: ***the queue size can only be set when creating a group***. User can use MCC_CreateGroupEx to change the queue size, the function prototype is as follow：

```
MCC_CreateGroupEx(  int xMapToCh,
                    int yMapToCh,
                    int zMapToCh,
                    int uMapToCh,
                    int vMapToCh,
                    int wMapToCh,
                    int aMapToCh,
                    int bMapToCh,
                    int nCardIndex,
                    int nMotionQueueSize);
```

nMotionQueueSize : Define the motion command queue size by user

The following is a list of commands that manipulate the motion command queue. By calling these commands, the MCCL will put a command into the motion command queue, get the first command in the queue (as well as remove it) at the appropriate time, and perform the corresponding action:

A. Linear motion command

MCC_Line()

B. Arc motion command

MCC_ArcXYZ()	MCC_ArcXYZ_Aux()
MCC_ArcXY()	MCC_ArcXY_Aux ()
MCC_ArcYZ()	MCC_ArcYZ_Aux ()
MCC_ArcZX()	MCC_ArcZX_Aux ()
MCC_ArcThetaXY()	MCC_ArcThetaYZ()
MCC_ArcThetaZX()	

C. Circular motion command

MCC_CircleXY()	MCC_CircleYZ()
MCC_CircleZX()	
MCC_CircleXY_Aux ()	MCC_CircleYZ_Aux ()
MCC_CircleZX_Aux ()	

D. Helical motion command

MCC_HelicaXY_Z()	MCC_HelicaYZ_X()
MCC_HelicaZX_Y()	
MCC_HelicaXY_Z_Aux ()	MCC_HelicaYZ_X_Aux ()
MCC_HelicaZX_Y_Aux ()	

E. Point-to-Point motion command

MCC_PtP()	MCC_PtPX()	MCC_PtPY()
MCC_PtPZ()	MCC_PtPU()	MCC_PtPV()
MCC_PtPW()	MCC_PtPA()	MCC_PtPB()

F. Jog motion command

Jog motion (Unit: Pulse) / Jog motion (Unit: User Unit, UU (the unit is determined by the user)) / Continuous jog motion (Unit: User Unit, UU)

MCC_JogSpace()   MCC_JogConti()   MCC_JogPulse()

G. In Position command

MCC_EnableInPos()   MCC_DisableInPos()

H. Path blending command

MCC_EnableBlend()   MCC_DisableBlend()   MCC_CheckBlend()

I. Delay motion command

MCC_DelayMotion()

Usage of the above commands when the motion command queue is full will result in a return value of COMMAND_BUFFER_FULL_ERR, meaning that the command cannot be accepted. Fig. 2.3.1 represents the operational process for Group 0 motion command queue and demonstrates that motion commands of the same group will be executed sequentially. Since each group has its exclusive motion command queue, motion commands from different groups can be executed simultaneously.

CAUTION: Commands that are not in **A** through **I** of section **2.3** will not be queued, and be executed immediately.

For example: setting X axis of Group 0 move to the coordinate 10, output the servo-on signal and have this axis move to the coordinate 20. The program can be written as follows:

```
MCC_Line(10, 0, 0, 0, 0, 0, 0, 0, 0);
```

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

Then, once MCC_Line() has been placed in the motion command queue (not executed yet), MCC_SetServoOn() will be executed immediately. Because MCC_SetServoOn() is executed immediately instead of being placed in the queue, the servo-on signal will be sent before the actual position reaches coordinate 10. The user needs to pay extra attention to this operational characteristic.

If the signal output is required to execute after X axis reaches coordinate 10, the user needs to make the additional determination. Users need to verify system motion status or current coordinate by themselves to control the signal output. The following is a simple example:

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

        MCC_Line(10, 0, 0, 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, 0, 0);
```

2.4 Mechanism、Encoder and Go Home Parameter Settings

2.4.1 Mechanism Parameters

The MCCL uses mechanism parameters to define mechanical platform characteristics and driver usage type of the user as well as program the positioning system, boundary values of coordinate system, and the maximum safe feed speed of each axis corresponding to the working home position.

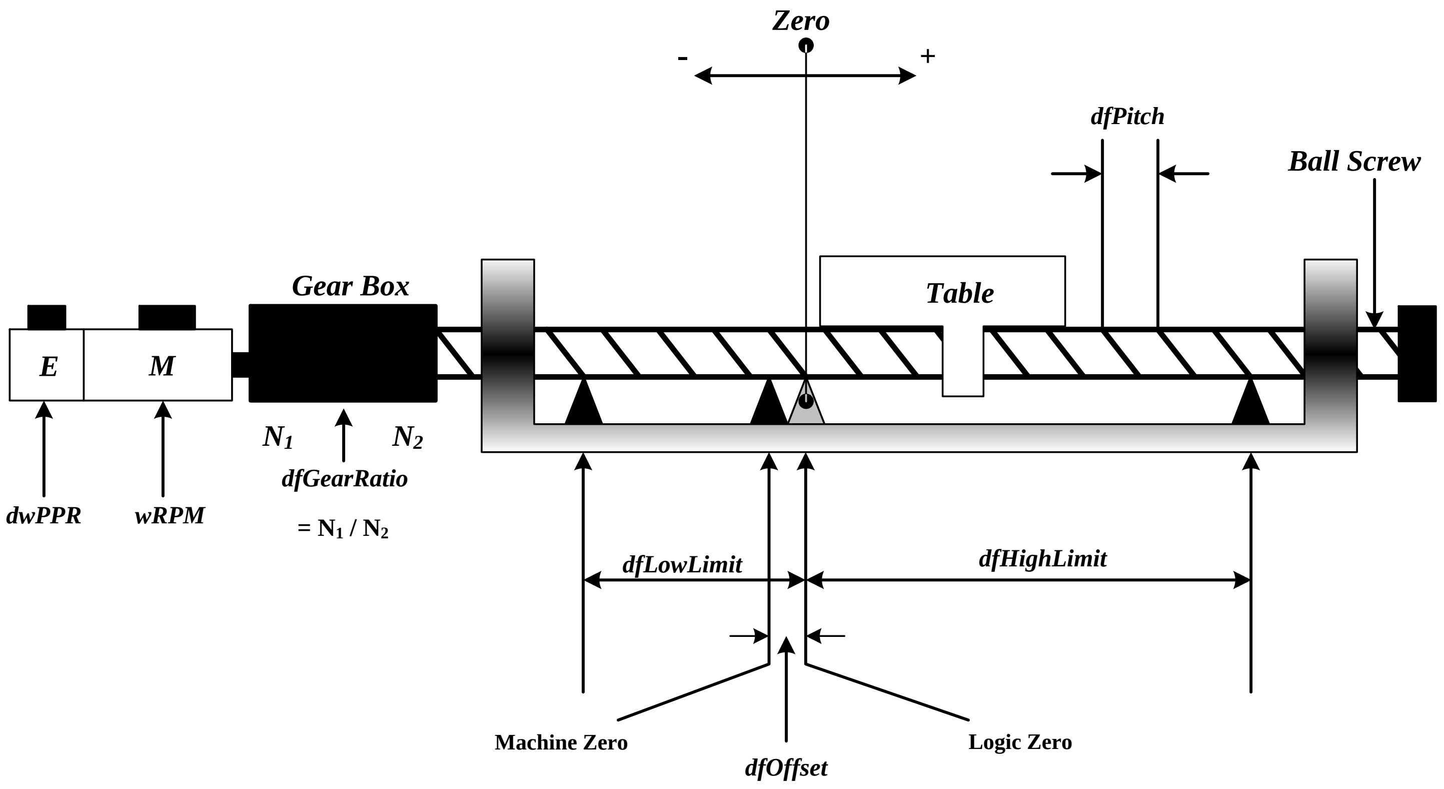


Fig. 2.4.1 Mechanical platform characteristics

Below is a detailed description of the mechanism 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       wOverTravelDownSensorMode;
} SYS_MAC_PARAM;
```

wPosToEncoderDir: Directional adjustment parameter

0	Output command does not reverse
1	Output command reverses

This parameter is used to correct the direction of motion command when it differs from the desired structural motion direction. If a forward motion command such as MCC_JogSpace(10, 10, 0, 0) is sent, but the machine actually moves in the direction opposite to user definition due to the motor wiring, this parameter can be set to “1” at this point to align the direction of motion command with the desired motion direction. (altering motor wiring is not required).

wRPM: Maximum speed of motor rotations

The maximum speed of motor rotations for each axis. **When conducting fast point-to-point motion**, the motor speed of each axis converted from the speed setting will not exceed the *wRPM* setting.

➔ *See Also* MCC_SetPtPSpeed()

dwPPR: Increases the encoder count for each revolution of the motor or requires pulses per rotation.

When closed loop control is used, this value is the increase of encoder count with every revolution of the motor; if an open circuit system is used, then this value is

pulses required per revolution.

When using a linear motor, both *dfPitch* and *dfGearRatio* should be set to 1. Additionally, a linear motor does not have definitions related to *dwPPR* and pulse is the unit of meaure for distance

So *dwPPR* can be set as 1 at this point to change the unit used in the MCCL into a pulse. For example, when X-axis is required to move 1000 pulses, MCC_Line(1000, 0, 0, 0, 0, 0, 0, 0, 0) can be called for the X axis to output 1000 pulses; when MCC_SetFeedSpeed(500) is used, it means that the required linear motor speed is 500 pulses per second.

dfPitch: Ball screw backlash

The distance which the table moves for each revolution of the ball screw; unit: UU. If there is no ball screw configuration, this value should be set to 1.

dfGearRatio: Gearbox deceleration ratio

The two-way gear ratio of the gearbox connecting the motor shaft and the ball screw; this value can be calculated by using the number of gear gaps or simply defined as "number of motor rotations per ball screw revolution". If there is no gearbox configuration, 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 working home position; unit: UU.

➔ *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 working home position and is often set as a negative value; unit: UU.

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 IMP does not perform this function)

wCommandMode: Motion command output mode

OCM_PULSE	Pulse Command
OCM_VOLTAGE	Voltage Command

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

wPaddle: To reserve the field, set to 0.

wOverTravelUpSensorMode: Positive limit switch wiring; please refer to the below description 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 on the axis indicated.

wOverTravelDownSensorMode: Negative limit switch wiring; please refer to the below description 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 on the axis indicated.

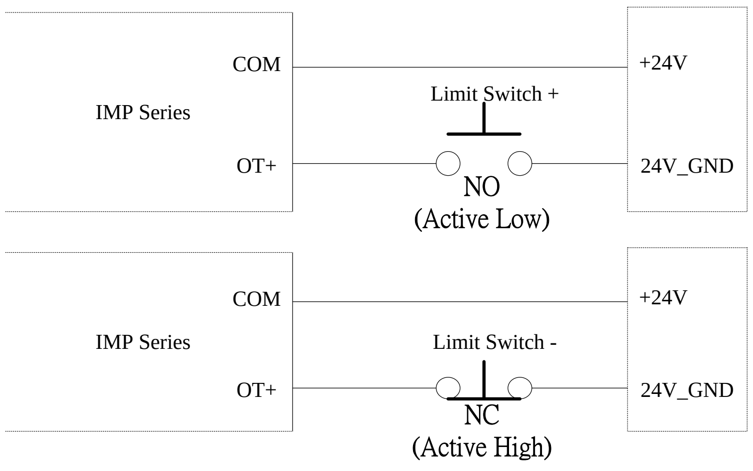


Fig. 2.4.2 Limit switch wiring

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 limit switch status 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 wiring setting; to fix this, *wOverTravelUpSensorMode, wOverTravelDownSensorMode* or both need to be changed.

In order to make the limit switch operate normally, it is also necessary to call MCC_EnableLimitSwitchCheck() so that the settings of *wOverTravelUpSensorMode* and *wOverTravelDownSensorMode* become effective.

However, if *wOverTravelUpSensorMode* and *wOverTravelDownSensorMode* are set as SL_UNUSED, then it is meaningless to call MCC_EnableLimitSwitchCheck().

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

MCC_EnableLimitSwitchCheck() is generally used in combination with MCC_GetErrorCode(). Continuous calling of MCC_GetErrorCode () can verify if the system has produced an error record because the limit switch is triggered (codes 0xF701 to 0xF708 represents the limit switch is triggered by axis X to B respectively). When an error from a triggered limit switch is discovered, the common response can be a message displayed on the screen to notify the operator. Then MCC_ClearError() is called in the program to clear the error so that the system can travel in the opposite direction to move away from the limit switch.

After the content of each field is confirmed, the user can use MCC_SetMacParam() to set the mechanism parameters. Below is an example:

```
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; // any value
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
```

Generally, the mechanism parameters must be set before the MCC_InitSystem() is used. The mechanism parameter of each axis 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 type, signal input phase swap and counts per encoder cycle (×1, ×2, ×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 setting

0	Not inverse
1	Inverse

wBInverse: Phase B signal inverse setting

0	Not inverse
1	Inverse

wCInverse: Phase C (Phase Z) signal inverse setting

0	Not inverse
1	Inverse

wABSwap: Phase A/B signal swap setting

0	Not swap
1	Swap

wInputRate: Set counts per encoder cycle

1	1 counts per encoder cycle (x1)
2	2 counts per encoder cycle (x2)
4	4 counts per encoder cycle (x4)

wPaddle: To reserve the field, set to 0.

After each field content within the encoder parameters is confirmed, the encoder parameter can be set by 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 4 counts per encoder cycle

MCC_SetEncoderConfig(&stENCConfig, 0, 0); //   set Axis 0 in Card 0
```

The encoder parameters must be set before the MCC_InitSystem() is used. The encoder parameters of each axis must be set separately.

CAUTION

If the mechanism or encoder parameters are changed after MCC_InitSystem() is called, then MCC_UpdateParam() must also be called so that the system can respond to the new settings. However, ***the effect of using MCC_UpdateParam() is similar to the effect of using MCC_ResetMotion() and the system will return to the initial status 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 usage mode, go home motion direction, home sensor wiring, encoder index signal count, and acceleration and deceleration durations. For details of this function, please refer to section **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 parameter value must be greater than or equal to 3 and smaller than or equal to 16. For detailed descriptions of each mode, please refer to 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

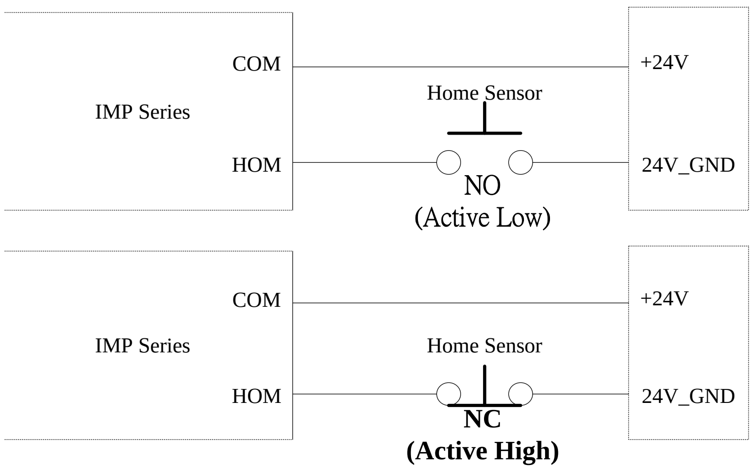


Fig. 2.4.3 Home sensor wiring

To use the go home function, *wSensorMode* must be correctly set according to the home sensor wiring (see Fig. 2.4.3). MCC_GetHomeSensorStatus() can be used to verify the home sensor status of the wiring settings. If the home sensor status obtained by using MCC_ GetHomeSensorStatus() is active when the home sensor is not triggered, this means the wiring setting is incorrect and the setting *wSensorMode* must be changed.

***wPaddle0*:** To reserve the field, set to 0.

nIndexCount: Indicated encoder index signal number

For phase 2 in go home motion process (searching for the indicated index number), the code for the first index signal occurs is 0, the one for the second index signal is 1 and follow this order for the subsequent index signals. For some go home modes, it is necessary to indicate the encoder index signal number. When the index signal satisfies this setting is triggered, the entire go home motion can be completed.

wPaddle1: To reserve the field, set to 0.

dfAccTime: The time required to accelerate to *dfHighSpeed* or *dfLowSpeed* during the go home procedure. Unit: ms.

dfDecTime: The time required to decelerate from *dfHighSpeed* or *dfLowSpeed to stop* during the go home motion. Unit: ms.

dfHighSpeed: High speed setting. Unit: UU/sec.
This parameter is the speed used during the first phase of the go home procedure.

dfLowSpeed: Low speed setting. Unit: UU/sec.
This parameter is the speed used during the final phase of completing the go home motion.

dfOffset: Distance between the working home position and machine home position. Unit: UU.

Generally, the displacement between the machine home position and the working home position will be found during calibration. To confirm this displacement, first set *dfOffset* as 0. When the go home procedure is completed (the platform stops at the "machine home position"), use the jog command to move the platform to the "working home position" and use this displacement to set *dfOffset*. After performing go home procedure again, the motion axis will move to the "working home position" and the system will use this position as the reference home of motion command.

After the content of each field of go home parameters is confirmed, the user can use MCC_SetHomeConfig() to set go home parameters. Below is an example:

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

stHomeConfig.wMode          = 3;   // use 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; unit: ms
stHomeConfig.dfDecTime          = 300;    // time required for deceleration; unit: ms
stHomeConfig.dfHighSpeed        = 10;     // unit: UU/sec
stHomeConfig.dfLowSpeed         = 2;      // unit: UU/sec
stHomeConfig.dfOffset           = 0;

MCC_SetHomeConfig(&stHomeConfig, 0, 0); //    set 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 created before using the MCCL. A group can be considered as an independent motion system. When this system moves, an interdependent relationship often exists between each internal motion axis. The X-Y-Z platform is an example of this.

The MCCL implemented in a group operation concept and most of the motion control commands are operated in groups. Each group consists of eight motion axes: X, Y, Z, U, V, W, A, and B; each motion axis is not required to actually correspond to an output channel on IMP. The MCCL can simultaneously control up to six IMPs while each card can define up to eight groups. Therefore, a maximum of 48 independent groups can be used at the same time without affecting the operation of each other.

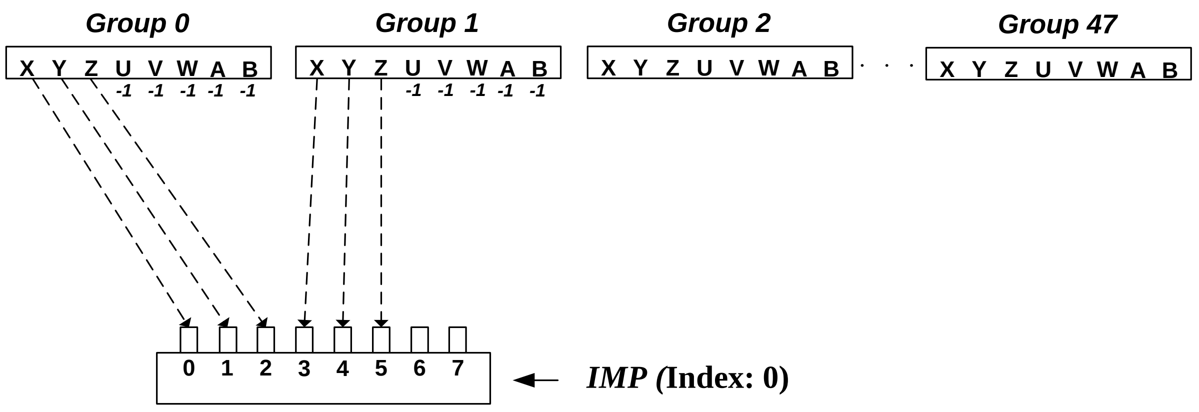


Fig. 2.4.4 Group parameter settings

Using Fig. 2.4.4 as an example, two groups and one IMP are used. The results of the trajectory planning for the X, Y, and Z axes in Group 0 are respectively output from the physical output channels 0, 1, and 2 in Card 0 while the result of trajectory planning for the U, V, W, A, and B axes is ignored. The result of trajectory planning for the X, Y, and Z axes in Group 1 are respectively output from the output channels 3, 4, and 5 in Card 0 while the result of trajectory planning for the U, V, W, A, and B axes is ignored.

The program can be written as follows:

```
int    nGroup0, nGroup1;

MCC_CloseAllGroups(); // disable all groups
nGroup0 = MCC_CreateGroup(    0,   // X corresponds to a physical output
                                   channel 0
                              1,   // Y corresponds to a physical output
                                   channel 1
                              2,   // Z corresponds to a physical output
                                   channel 2
                              -1,  // U does not correspond to a physical
                                   output channel
                              -1,  // V does not correspond to a physical
                                   output channel
```

```
                                  -1,  // W does not correspond to a physical
                                           output channel
                                  -1,  // A does not correspond to a physical
                                           output channel
                                  -1,  // B does not correspond to a physical
                                           output channel
                                  0);  // corresponds to Card 0

    nGroup1 = MCC_CreateGroup(    3,   // X corresponds to a physical output
                                           channel 3
                                  4,   // Y corresponds to a physical output
                                           channel 4
                                  5,   // Z corresponds to  a physical output
                                           channel 5

                                  -1,  // U does not correspond to a physical
                                           output channel
                                  -1,  // V does not correspond to a physical
                                           output channel
                                  -1,  // W does not correspond to a physical
                                           output channel
                                  -1,  // A does not correspond to a physical
                                           output channel
                                  -1,  // B does not correspond to a physical
                                           output channel
                                  0);  // corresponds to Card 0
```

The return value of MCC_CreateGroup() represents the newly established group number (0 to 47). This group number will be used later when calling motion commands. For example, when the user wants to move axes X, Y, and Z in Group 1 to coordinate 10, the program should be written as MCC_Line(10, 10, 10, 0, 0, 0, 0, 0, ***nGroup1***); channels 3, 4, and 5 on IMP Card 0 will be responsible to 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, 0, 0, ***nGroup0***); // Command 0

MCC_Line(20, 20, 20, 0, 0, 0, 0, 0, ***nGroup0***); // Command 1

MCC_Line(10, 10, 10, 0, 0, 0, 0, 0, ***nGroup1***); // Command 2

MCC_Line(20, 20, 20, 0, 0, 0, 0, 0, ***nGroup1***); // Command 3

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

Since each group operates independently, Group 1 does not need to wait until Group 0 finishes Command 0 before directly executing Command 2 as well as outputting the trajectory planning for axes X, Y, and Z from channels 3, 4, and 5 in Card 0. After Command 2 is completed, Group 1 will then execute Command 3 from the same group.

If no groups are created before activating the MCCL, then the MCCL will operate by default. The default operation simply enables the index to be Group 0 and its motion axes X, Y, Z, U, V, W, A, and B to correspond to output channels 0 to 7 on Card 0.

Note

1. Groups **do not** affect each other.
2. Groups all contain 8 motion axes (X, Y, Z, U, V, W, A, and B) that can be programmed to correspond to a physical output channel or not. However, **at least one motion axis in the group should be correspond to a physical output channel. In addition, two motion axes cannot correspond to the same physical output channel.**
3. To reduce CPU usage rate, **minimize the number of groups used**.

2.5 Initialize and Close the Motion Control Command Library

2.5.1 Initializing the Motion Control Command Library

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

a.	Mechanism 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. Please refer to the previous description and "**IMP Series Motion Control Command Library Example Manual**" for settings of the machine, the encoder and the **Group** (motion group). The following will only introduce how to enable the MCCL.

I. Initializing the MCCL

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

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

nInterpolateTime is the interpolation time (please refer to the explanation in a later section); unit: ms. The setting limits are between 1 ms to 50 ms, with a suggested value of 2 ms. Shorter interpolation times will reduce the distance between two interpolation points, but increase the workload of CPU. Below is a reference to interpolation time settings. These suggested values are not absolute and should be adjusted according to actual needs.

System Characteristics	Suggested Interpolation Time
Only requires linear motion	5 ms ~ 10 ms
Generally includes arc motion	5 ms
Required arc motion trajectories to be circular	1ms ~ 3ms

pstCardConfig is the IMP hardware parameter settings for the previous step. *wCardNo* is the number of IMP used at this time.

2.5.2 Close the Motion Control Command Library

To close the MCCL, simply call MCC_CloseSystem() command.

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. Acquire current position coordinates

➔ *See Also* MCC_GetCurPos() MCC_GetCurRefPos()
MCC_GetPulsePos()

III. Enable/disable software over-travel check function

Once MCC_SetOverTravelCheck() is used to enable the over-travel check function, after calculating each interpolation point, the MCCL will check if the interpolation point exceeds the effective work zone of each axis. If the point is verified to exceed the work zone, commands will not be sent to the motion control card. The user can use MCC_GetErrorCode() to check the information mode (for the meaning of information code, please refer to IMP Series Motion Control Command Library Reference Manual) to confirm if the effective work zone has been exceeded.

➔ *See Also* MCC_GetOverTravelCheck()
MCC_GetErrorCode()

IV. Enable/disable hardware limit switch check function

For this function, please refer to the description in section **"2.4.1Mechanism Parameters"**.

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

2.6.2 Basic Trajectory Planning

The MCCL provide linear, arc, circular and helical motion (collectively referred to as general motion) along with the point-to-point motion trajectory planning. Before using these functions, it is required to set acceleration and deceleration types (S or trapezoid), acceleration and deceleration durations and feed speed corresponding to mechanism characteristics and special requirements.

I. General Motion (Linear, arc, circular, and helical motion)

The general motion includes multi-axis synchronized motions such as linear, arc, circular and helical motions. Usually, the return value of these functions is checked when using the general motion command. If the return value is smaller than 0, it means the motion command is rejected; for reasons of rejection, please refer to the manual related to return value definitions (refer to "**IMP Series Motion Control Command Library Reference Manual**"). If the return value is larger than or equal to 0, the value is the index number given by the MCCL to the motion command. The user can follow the motion command execution process by these index numbers. MCC_ResetCommandIndex() can be used to reset this index value to restart counting from 0.

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 and deceleration durations are 300 ms.

➔ *See Also* MCC_Line()

B. Arc Motion

When calling the arc motion command, only the reference and end point coordinates need to be set. Based on the feed speed motion provided, the preset acceleration and deceleration durations are 300 ms. **The MCCL also provides 3-D arc motion command.**

➔ *See Also*	MCC_ArcXYZ()	MCC_ArcXYZ_Aux()
	MCC_ArcXY()	MCC_ArcXY_Aux ()
	MCC_ArcYZ()	MCC_ArcYZ_Aux ()
	MCC_ArcZX()	MCC_ArcZX_Aux ()
	MCC_ArcThetaXY()	MCC_ArcThetaYZ()
	MCC_ArcThetaZX()	

C. Circular Motion

When calling the circular motion command, only the center position 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 and deceleration durations are 300 ms.

➔ *See Also*	MCC_CircleXY()	MCC_CircleXY_Aux ()
	MCC_CircleYZ()	MCC_CircleYZ_Aux ()
	MCC_CircleZX()	MCC_CircleZX_Aux ()

D. Helical Motion

When calling the helical motion command, only the center position and the linear feed axis end point 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 and deceleration durations are 300 ms.

➔ *See Also*	MCC_HelicaXY_Z()
	MCC_HelicaYZ_X()
	MCC_HelicaZX_Y()
	MCC_HelicaXY_Z_Aux ()
	MCC_HelicaYZ_X_Aux ()
	MCC_HelicaZX_Y_Aux ()

E. General Motion Acceleration/Deceleration Time and Feed Speed

MCC_SetAccTime() and MCC_SetDecTime() can be used to set the desired general motion acceleration and deceleration durations; MCC_SetFeedSpeed() can be used to set the desired feed speed. Additionally, please note that the MCCL only considers the 3 axes X/Y/Z when calculating the feed speed of general motion while axes U/V/W/A/B simply start and end motion simultaneously aligning with the previous 3 axes for linear motion). If there is no displacement of X/Y/Z in this motion command, the set feed speed will be altered to indicate the speed of axes U/V/W/A/B that has traveled the furthest and the other 4 axes simultaneously start and end in concert (similar to point-to-point motion).

The feed speed setting cannot exceed the limit set by using MCC_SetSysMaxSpeed(); the value set by MCC_SetSysMaxSpeed() will become the feed speed when the limit is exceeded. 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 the linear motion in general motion. The only different is that the speed of general motion is set by MCC_SetFeedSpeed() in the unit of UU/sec while point-to-point motion uses the maximum safe feed ***ratio*** with the corresponding command MCC_SetPtPSpeed(). This ratio is calculated as follows:

point-to-point feed speed for each axis =

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

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

Once the feed speed of 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 other axes starting and ending simultaneously.

Point-to-point acceleration and deceleration durations still follows the settings in general motion.

➔ *See Also*	MCC_PtP()
	MCC_GetPtPSpeed()

III. Jog motion (Unit: Pulse) /Jog motion (Unit: UU) / Continuous jog motion (Unit: UU)

A. **Jog motion (Unit: Pulse)**: 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 of MCC_GetMotionStatus() should be GMS_STOP).

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

B. **Jog motion (Unit：UU)**: MCC_JogSpace()

This command requires a specific axis to move by the indicated displacement (Unit: UU) according to the indicated feed speed ratio (please see the description of point-to-point motion). When using this command, motion status must be stopped (the return value of MCC_GetMotionStatus() should be GMS_STOP). MCC_AbortMotionEx() can be used to stop this motion. The following is an example of using this command.

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

C. **Continuous jog motion (Unit：UU)**: MCC_JogConti()

This command requires the selected axis to move according to the indicated feed speed ratio (please see the description of 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 mechanism parameters). When using this command, motion status must be stopped (the return value of MCC_GetMotionStatus() should be GMS_STOP). MCC_AbortMotionEx() can be used to stop this motion. The following is an example of using this command.

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

IV. Pause, Continue and Abort Motion

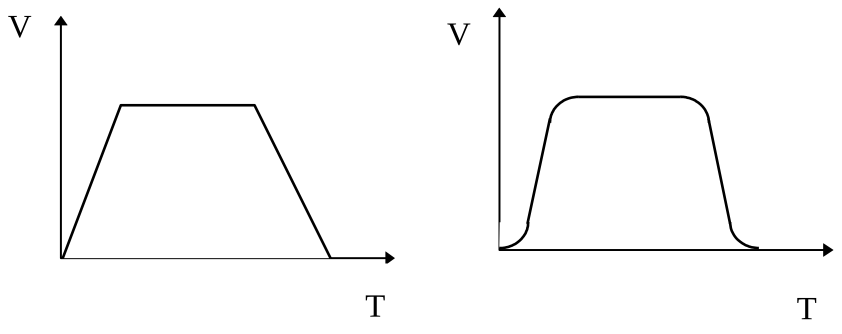
MCC_AbortMotionEx() can be used to abort all motion commands currently being executed and stored. MCC_HoldMotion() can be used to pause the motion command being executed (the motion is constantly decelerated to a stop at this point). At this time, the system will only continue executing the unfinished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

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

➔ *See Also* MCC_GetMotionStatus()

2.6.3 Advanced Trajectory Planning

To achieve more flexible and effective in position control, the MCCL provides several advanced trajectory planning functions. For example, motion blending function can be used when the precise positioning is not required between different motion commands while it is necessary to reach a designated position quickly. For common tracking problems in the control system, the MCCL also provides override speed function that allows dynamic adjustment of feed speed. Below are descriptions for each of these functions.



Trapezoid Curve S Curve

Fig. 2.6.1 Acceleration and deceleration types

I. Acceleration and Deceleration Types Settings

Acceleration and deceleration types can be set as either a trapezoid curve or an S curve (see Fig. 2.6.1). The type used for each axis of point-to-point, linear, arc, circular, helical motion can be set using the identical approach.

➔ *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() can be used to enable motion blending function. This function can satisfy the requirement to achieve a continuous blending in speed between different motion commands (meaning that before completing the previous motion command, it does not need to decelerate to stop and can directly accelerate or decelerate to the speed required by the next motion command). The motion blending functions include linear-linear, linear-arc, and arc-arc motion blending.

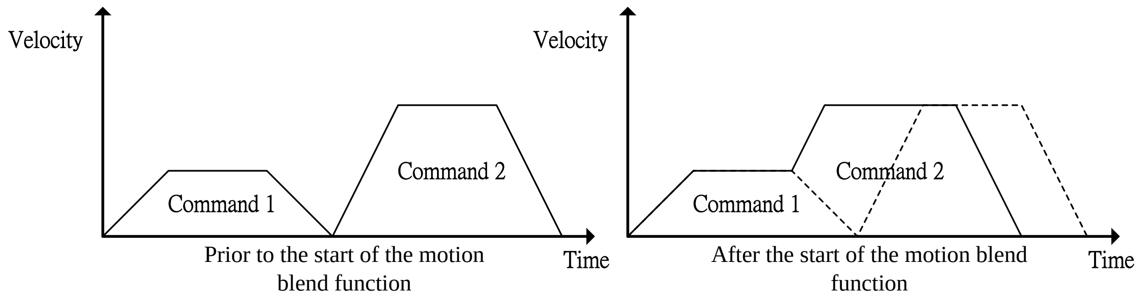


Fig. 2.6.2 Speed during motion blend

Fig. 2.6.2 shows the motion status after motion blending is enabled. The first motion command directly accelerates to the stable speed of the second motion command from its own stable speed without decelerating (as the solid line in Fig. 2.6.2 shows). With this method, the command is executed faster while the trajectory error will exist at the connection points between each command. Fig. 2.6.3 show the motion trajectory after motion blending is enabled (the dotted line show the originally planned trajectory curve).

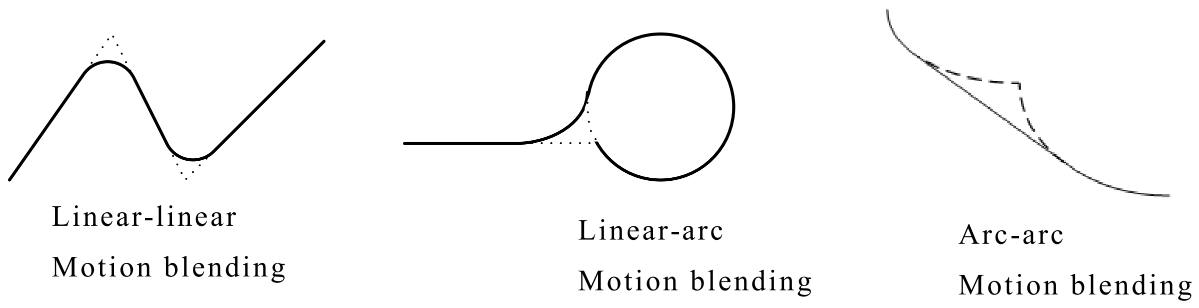


Fig. 2.6.3 Linear-linear, linear-arc, arc-arc motion blending

➔ *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 command being executed V1 to the required speed V2 (when V1 < V2), or decelerate from the current speed V3 to the required speed V4 (when V3 > V4).

In Fig. 2.6.4, V2 = V1 × 175 / 100 (using MCC_OverrideSpeed(175)); similarly, V4 = V3 × 50 / 100 (using MCC_OverrideSpeed(50)).

Using the speed ratio indicated by MCC_OverrideSpeed() to change tangential speed instantaneously and forcibly. 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()

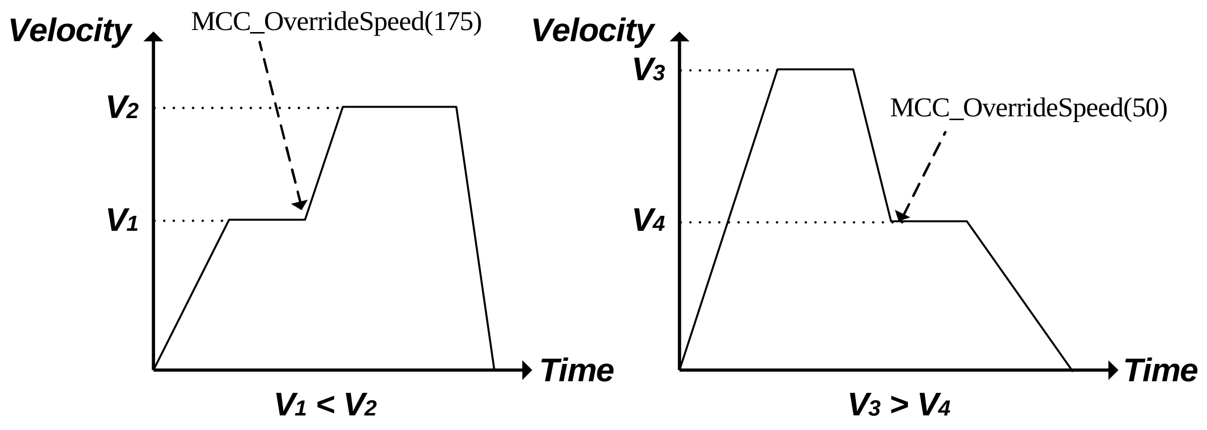


Fig. 2.6.4 Override speed

Point-to-Point Motion Override Speed:

MCC_OverridePtPSpeed() forcibly and instantaneously 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 and multiplied by 100. Please refer to the previous description. Using MCC_OverridePtSpeed() 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 obtain the motion path in advance, 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() forcibly delays the execution of the next motion command. The unit of the delay time is ms; an example is displayed below:

```
MCC_Line(10, 10, 10, 0, 0, 0, 0, 0, 1);    -------- A
MCC_DelayMotion(200, 1);
MCC_Line(15, 15, 15, 0, 0, 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 conditions such as *motion over travel* (the motion exceeds software boundary), the feed speed is greater than the maximum set value, the acceleration or deceleration speed is greater than the maximum set value, arc command parameter error or arc command execution error occurs, MCC_GetErrorCode() can be used to acquire the content of error code (for the meaning of error code, please refer to ”**IMP Series Motion Control Command Library Reference Manual**”).

When an error occurs in a group, this group will not execute another motion command. Meanwhile, the user must manually use MCC_GetErrorCode() to identify the reason for the error as well as solve it. MCC_ClearError() can then be used to clear the error history and return the group to normal status.

2.6.4 Interpolation Time and Acceleration/Deceleration Time

I. Interpolation Time Setting

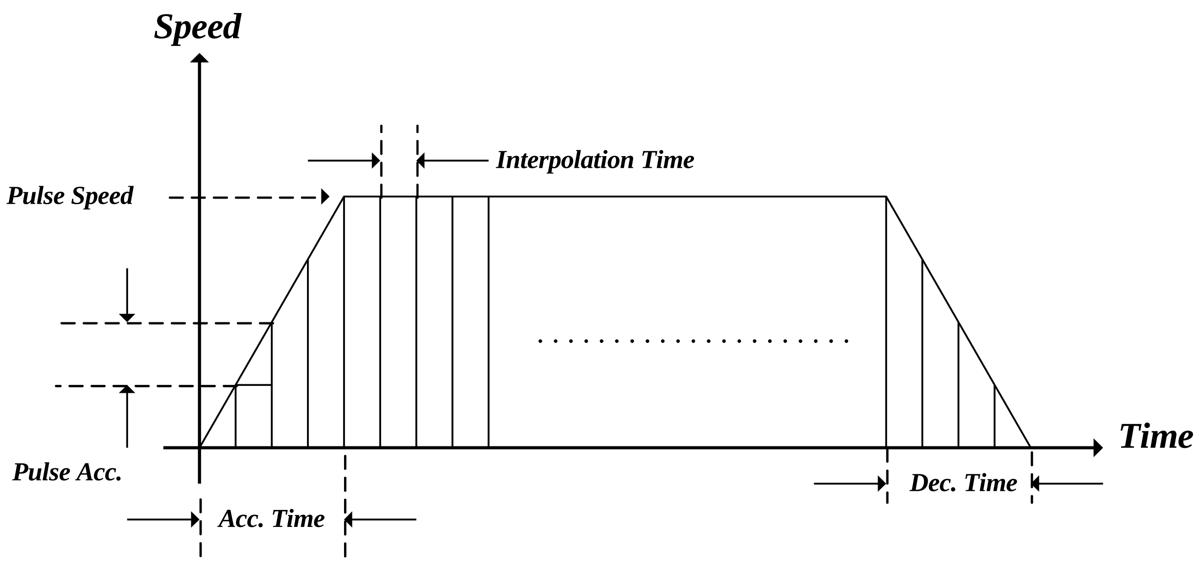


Fig. 2.6.5 Trajectory planning parameters

Interpolation time is the time gap to the next interpolation point (see Fig. 2.6.5). The minimum setting is 1 ms and the maximum is 50 ms.

II. Maximum pulse speed setting

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

➔ *See Also* MCC_GetMaxPulseSpeed()

III. Maximum Pulse limits Acceleration and Deceleration speed

The acceleration and deceleration durations are insufficient during the motion process, and the acceleration and deceleration speed may exceed the value allowed by the machine. Hence, this may damage the machine because of excessive motion inertia. This setting can be used to limit the sent pulse difference within the tolerable range of the machine. MCC_GetErrorCode() can determine whether acceleration or deceleration speed has exceeded the set range within interpolation times

MCC_SetMaxPulseAcc() sets the maximum pulse acceleration and deceleration speed between 1 to 32767. The default is 32767.

➔ *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 the time needed to decelerate from a stable speed to a stop. MCC_SetAccTime() and MCC_SetDecTime() set the acceleration and deceleration time required for linear, arc, circular and helical motion. MCC_SetPtPAccTime() and MCC_SetPtPDecTime() set the acceleration and deceleration time required for point-to-point motion. Faster feed speeds often need longer acceleration times. Therefore, MCC_SetAccTime() and MCC_SetDecTime() are often used in combination with MCC_SetFeedSpeed(). Similarly, MCC_SetPtPAccTime() and MCC_SetPtPDecTime() are often used in combination with MCC_SetPtPSpeed().

The following example explains the requirements for different acceleration and deceleration durations for different feed speed. Generally, users must customize the content of SetSpeed() according to mechanism characteristics. SetSpeed() should be used when it is necessary to change feed speed. To avoid the loss of steps, MCC_SetFeedSpeed() should not be called directly, especially when using a stepper motor.

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

        dfAcc = 0.04; // set acceleration to 0.04 (UU/sec^2)

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

                MCC_SetAccTime(dfTime);
                MCC_SetDecTime(dfTime);

                MCC_SetFeedSpeed(dfSpeed);
        }
}
```

2.6.5 System Status Check

Commands provided by the MCCL can verify current actual position, planned and actual speeds, motion status, motion command stock, FMC stock in hardware FIFO, and the content of motion command being executed.

MCC_GetCurPos() can be used to obtain current command position. Unit: UU.

MCC_GetPulsePos() can acquire the pulses sent from the control card. The only difference between this value and the one obtained by MCC_GetCurPos() is that the latter goes through mechanism parameter conversion.

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

MCC_GetPtPSpeed() can be used to acquire the feed speed ratio of point-to-point motion trajectories and MCC_GetFeedSpeed() can acquire the feed speed of general motion planning. For general motion, the user can also use MCC_GetCurFeedSpeed() to acquire current actual tangential speed and MCC_GetSpeed() to acquire current actual feed speed of each axis.

The return value obtained from calling MCC_GetMotionStatus() can verify the current motion status. If the return value is GMS_RUNNING, it means the system is in motion; if the value is GMS_STOP, it means the system has stopped and there is no stock command to be executed; if the value is GMS_HOLD, it means the system has been paused by using MCC_HoldMotion(); if the value is GMS_DELAYING, it means the system is currently delayed by using MCC_DelayMotion().

MCC_GetCurCommand() can be used to obtain the information related to the motion command currently being executed. The command declaration of MCC_GetCurCommand() is as follows:

```
    MCC_GetCurCommand(COMMAND_INFO *pstCurCommand,
                                          WORD wGroupIndex)
```

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

```
typedef struct _COMMAND_INFO
{
        int             nType;
        int             nCommandIndex;
        double          dfFeedSpeed;
        double          dfPos[8];
} COMMAND_INFO;
```

nType: Motion command type

0.	Point-to-Point Motion
1.	Linear motion
2.	Clockwise arc, or clockwise circular motion
3.	Counter-clockwise arc, or cunter-clockwise 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	current remaining delay time (unit: ms)

dfPos[]: Required destination position

MCC_GetCommandCount() can be used to obtain the motion command in the stock that has not been executed yet. This stock does not include the motion command currently being executed.

MCC_GetCurPulseStockCount() can be used to obtain the fine movement command (FMC) stock in the IMP. During continuous motion, the default FMC stock is 60. Users can set FMC stock to ensure stable motion performance. If FMC stock equals 0, it is necessary to extend interpolation time (please refer to the previous description related to interpolation time). In addition, extending of interpolation time should also be considered if the lag appears in the user interface display.

2.7 Position Control

The MCCL provides the following for position control functions:

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

The following sections will introduce the content and the usage of each function.

2.7.1 Closed Loop Proportional Integration Differentiation Forward Feed Gain (PID+FF Gain) Setting

Use MCC_SetPGain(), MCC_SetIGain(), MCC_SetDGain() and MCC_SetFGain() to set the PID+FF gain parameters in closed-loop control. The parameter can be set within the range between 0 to 255. The PID+FF gain parameters can be adjusted as follows: After the driver is configured as voltage command, use the [View Profile] in Integrated Test Environment (ITE) provided by the installation CD-ROM to adjust the gain according to the tracking error (the tracking error is the difference between the command position and the actual position).

➔ *See Also* MCC_GetPGain()
MCC_GetIGain()
MCC_GetDGain()
MCC_GetFGain()

2.7.2 In Position Confirmation

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

This function can be enabled by calling MCC_EnableInPos(). Once enabled, the

MCCL will start checking if the in position confirmation condition is met after sending the last FMC of that motion command. If it is in position, then the next motion command will be executed. However, if the command is still not in a position after the maximum check time set (use (MCC_SetInPosMaxCheckTime() to set) ends, the subsequent command will be aborted and an error record will be produced (refer to Fig. 2.7.1 for the definition of maximum check time).

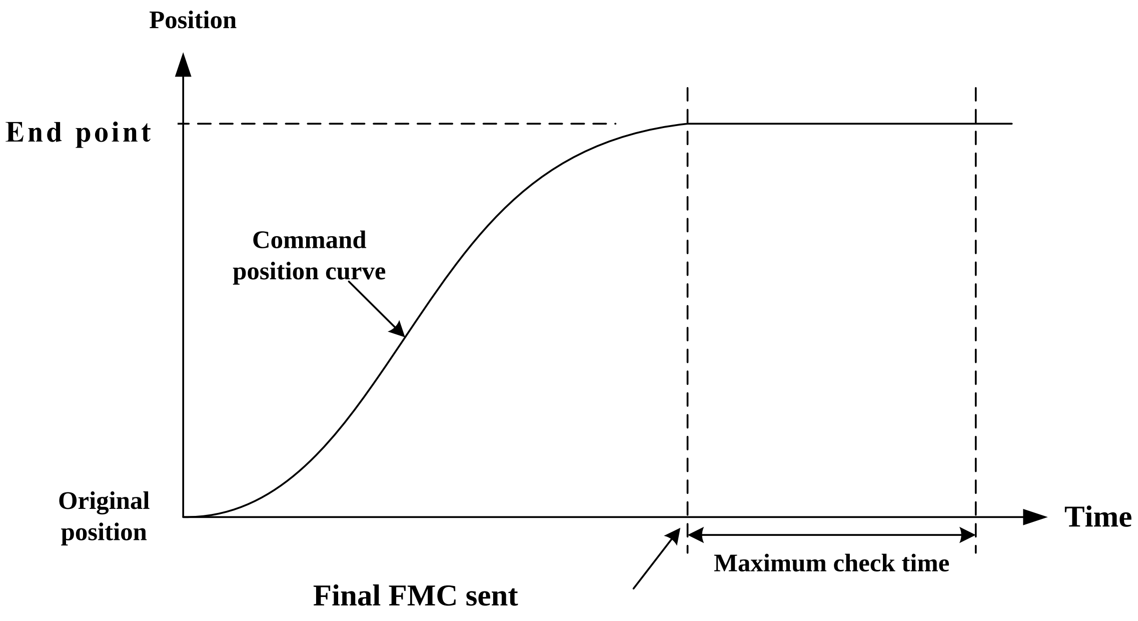


Fig. 2.7.1 Maximum check time diagram

The MCCL provides four types of in position confirmation mode. The user can select the appropriate type by using MCC_SetInPosMode() command. Definitions and descriptions of each mode are detailed as follows:

Mode IPM_ONETIME_BLOCK:

When the position error of each axis in the group is smaller than or equal to error tolerance range (can be set by using MCC_SetInPosToleranceEx(); unit: UU), the in position criteria of this mode are satisfied (see Fig. 2.7.2).

If the criteria have not been satisfied before the maximum check time ends, the subsequent command will be aborted and an error record will be produced (can obtain by MCC_GetErrorCode()).

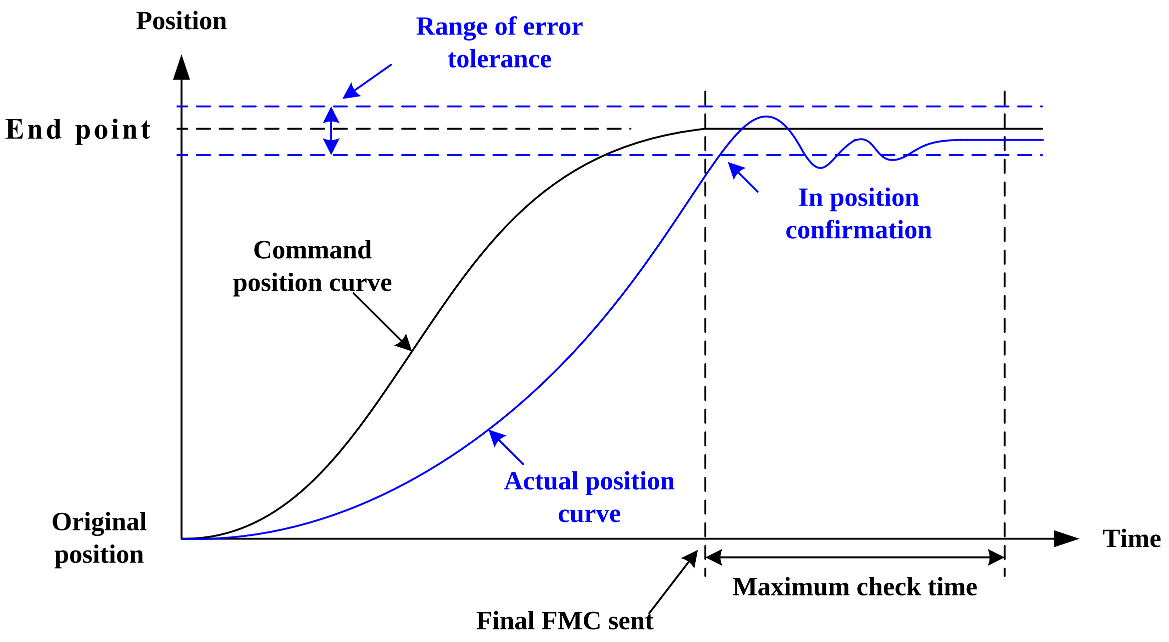


Fig. 2.7.2 IPM_ONETIME_BLOCK mode successful in position diagram

Mode IPM_ONETIME_UNBLOCK:

The in position criteria of this mode are identical to those of IPM_ONETIME_BLOCK. The only difference is that when the in position criteria have not been satisfied before the maximum check time ends, the subsequent command will be executed directly and an error record will not be produced.

Mode IPM_SETTLE _BLOCK:

When the position error of each axis in the group is smaller than or equal to error tolerance range (can be set by using MCC_SetInPosToleranceEx(); unit: UU) and ***remains a stable period*** (can be set by using MCC_SetInPosSettleTime(); unit: ms), the in position criteria of this mode are satisfied (see Fig. 2.7.3).

If the criteria have not been satisfied before the maximum check time ends, the subsequent command will be aborted and an error record will be produced (can obtain by MCC_GetErrorCode()).

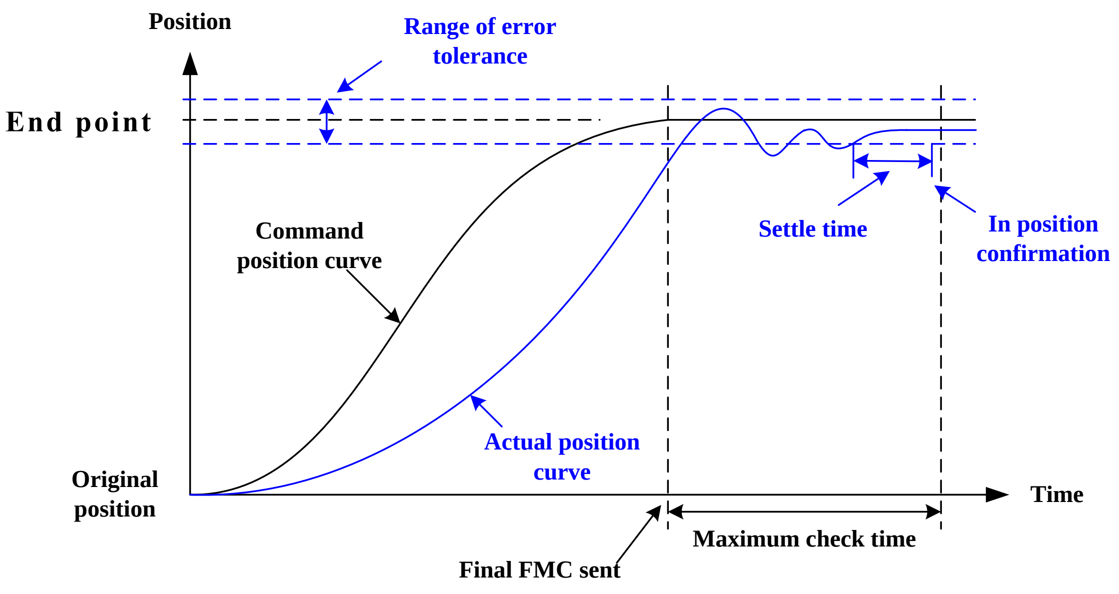


Fig. 2.7.3 IPM_SETTLE _BLOCK mode in position successful diagram

Mode IPM_SETTLE _UNBLOCK:

The in position criteria of this mode are identical to those of IPM_SETTLE_BLOCK. The only difference is that when the in position criteria have not been satisfied before the maximum check time ends, the subsequent command will be executed directly and an error record will not be produced.

The greater the in position tolerance error, the shorter the time required for completing in position confirmation. However, the error between the motion command connection point and the trajectory path will be greater (and will be smaller in the converse situation). As Fig. 2.7.4 shows, the smaller in position tolerance error will produce a more precise trajectory error (Error 1 < Error 2); hence, the in position tolerance error should be set appropriately according to different system functions. Meanwhile, MCC_GetInPosStatus() can be used to obtain the in position status of each motion axis in the group.

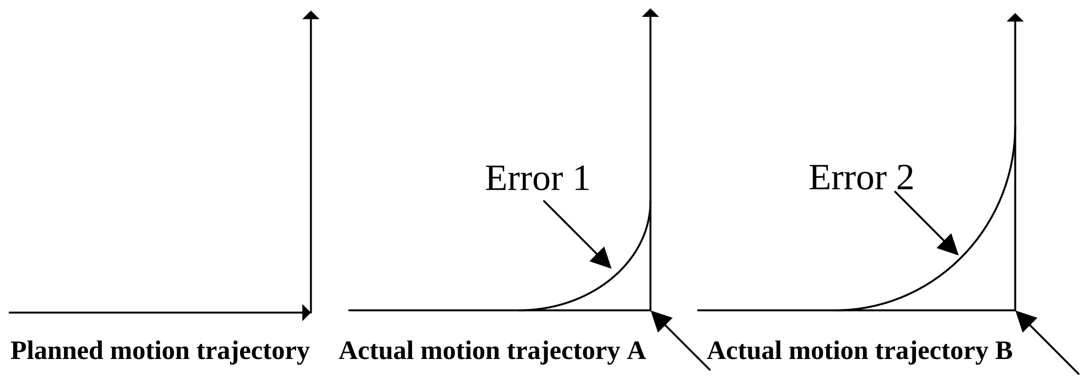

Error 1
Error 2
Planned motion trajectory
Actual motion trajectory A
Actual motion trajectory B

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

➔ *See Also* MCC_GetInPosToleranceEx()

MCC_DisableInPos()

CAUTION

1. Because the in position confirmation function is used to compare if the ***actual position*** and the target position are within the tolerance error range~~.~~ Remember verify the encoder connection.
2. Once the system has been verified as successfully the in position confirmation, further in position verification will not be used (meaning that it will hold the in position status even if the actual position leaves the in position tolerance range; see Fig. 2.7.2) until the new motion command is given.

2.7.3 Tracking Error

The error between the command position and actual position of the motion axis at any given moment is called as the tracking error (see Fig. 2.7.5).

Generally, the tracking error size is related to settings such as mechanism characteristics, closed-loop proportional gain and motion acceleration. An excessively large tracking error means the motion has deviated from (or lagged) the trajectory path too much and may even have had a collision.

To use MCC_SetTrackErrorLimit() to set error tolerance range and then use MCC_EnableTrackError() to activate this function. After the function has been enabled, once the tracking error of motion axis exceeds the range, the subsequent command trajectory of this group will be stopped and an error record will be produced. MCC_GetErrorCode() can be used to acquire the error code: 0xF801 to 0xF808 respectively shows the excessively large tracking error of X, Y, Z, U, V, W, A and B.

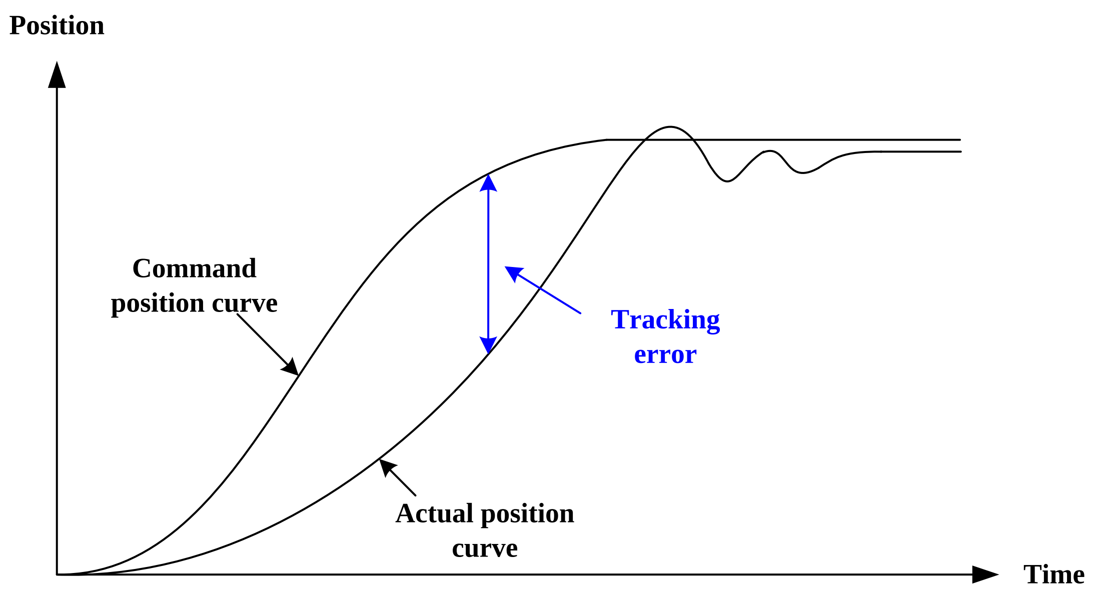


Fig. 2.7.5 Tracking error diagram

➔ *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. Refer to the following procedure to handle this:

1. Determine error type and conduct the corresponding solution (users should manually define this).
2. Call MCC_ClearError() to clear the error record.
3. The system continues normal operation.

2.7.4 Handling Positional Closed Loop Control Failure

When the closed-loop position control function fails because the proportional gain parameter is incorrectly set or for other operational reasons, the system will be in a non-controlled state. The user can set the value of allowable positive/negative pulse error by MCC_SetErrorCountThreshold(). When the system error register is larger than the setting, it will promptly alert the user that the system is in a non-controlled state and the motion control card will automatically produce an interruption signal. The user can customize a routine that interrupts closed-loop position control and serially connects to the system. This customized routine will be called when the motion axis closed-loop position control function fails, and the user can design the handling procedure into this customized routine. The procedures to use this function is detailed as follows:

***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)**

For example, the customized command can be designed as follows:

```
_stdcall MyPCLFunction(PCLINT *pstINTSource)
{

    //  determine whether the routine was triggered when the position error register is
greater than the set positive error in Channel 0
    if (pstINTSource->OVP0)
    {
     //  handling procedure of closed-loop position control function failure in
Channel 0
    }
    //  determine whether the routine was triggered when the position error register is
smaller than the set negative error in Channel 0
    if (pstINTSource->OVN0)
    {
     //  handling procedure of position closed loop control function failure in Channel
0
    }

    //  determine whether the routine was triggered when the  position error register
is greater than the set positive error in Channel 1

    if (pstINTSource->OVP1)
    {
     //  handling procedure of closed-loop position control function failure in
Channel 1
    }

    //  determine whether the routine was triggered when the positional error register
is smaller than the negative error setting in Channel 1
    if (pstINTSource->OVN1)
    {
     //  handling procedure of closed-loop position control function failure in
Channel 1
    }

    //  determine whether the routine was triggered when the  positional error
register is greater than the set positive error in Channel 2
    if (pstINTSource->OVP2)
    {
     //  handling procedure of closed-loop position control function failure in
Channel 2
    }
```

// determine whether the routine was triggered when the position error register is smaller than the set negative error in Channel 2

if (pstINTSource->*OVN2*)

{

// handling procedure of position closed loop control function failure in Channel 2

}

// determine whether the routine was triggered when the positional error register is greater than the set positive error in Channel 3

if (pstINTSource->*OVP3*)

{

// handling procedure of position closed loop control function failure in Channel 3

}

// determine whether the routine was triggered when the positional error register is smaller than the set negative error in Channel 3

if (pstINTSource->*OVN3*)

{

// handling procedure of position closed loop control function failure in Channel 3

}

A routine such as "else if (pstINTSource->*OVP1*)" cannot be used, because pstINTSource->*OVP0* and pstINTSource->*OVP1* may not be 0 simultaneously.

Later, use MCC_SetPCLRoutine (MyPCLFunction) to serially connect the customized ISR. When the customized is triggered during execution, the system can use the pstINTSource parameter declared as *PCLINT* in the customized routine to determine which trigger criterion was satisfied when the customized routine was called. The definition of *PCLINT* is as follows:

```
typedef struct _PCL_INT
{
        BYTE OVP0;
        BYTE OVP1;
        BYTE OVP2;
        BYTE OVP3;
        BYTE OVP4;
        BYTE OVP5;
        BYTE OVP6;
        BYTE OVP7;
        BYTE OVN0;
        BYTE OVN1;
        BYTE OVN2;
        BYTE OVN3;
        BYTE OVN4;
        BYTE OVN5;
        BYTE OVN6;
        BYTE OVN7;

} PCLINT;
```

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

OVP0	Channel 0 position error register is greater than the set positive error
OVP1	Channel 1 position error register is greater than the set positive error
OVP2	Channel 2 position error register is greater than the set positive error
OVP3	Channel 3 position error register is greater than the set positive error
OVP4	Channel 4 position error register is greater than the set positive error
OVP5	Channel 5 position error register is greater than the set positive error
OVP6	Channel 6 position error register is greater than the set positive error
OVP7	Channel 7 position error register is greater than the set positive error

OVN0	Channel 0 position error register is smaller than the set negative error
OVN1	Channel 1 position error register is smaller than the set negative error
OVN2	Channel 2 position error register is smaller than the set negative error
OVN3	Channel 3 position error register is smaller than the set negative error
OVN4	Channel 4 position error register is smaller than the set negative error
OVN5	Channel 5 position error register is smaller than the set negative error
OVN6	Channel 6 position error register is smaller than the set negative error
OVN7	Channel 7 position error register is smaller than the set negative error

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 movements, such as pitch or backlash error (see Fig. 2.7.6).

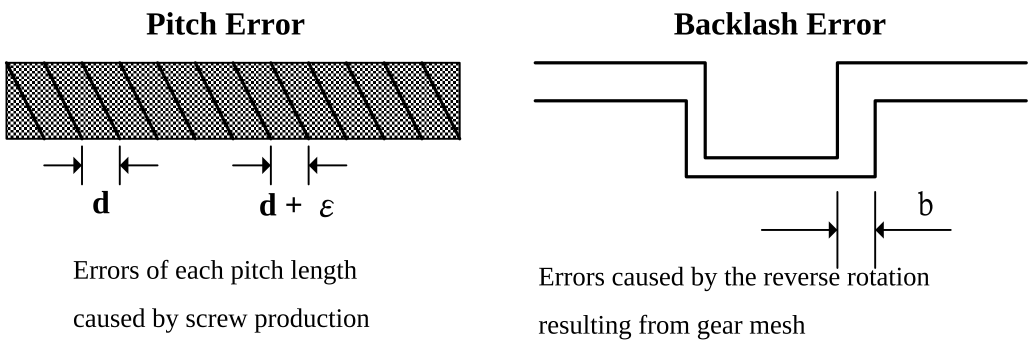

Pitch Error
d
d + ε
Errors of each pitch length
caused by screw production
Backlash Error
b
Errors caused by the reverse rotation
resulting from gear mesh

Fig. 2.7.6 Pitch error 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; record the number of segment error to make a forward and backward compensation table. This compensation table is a two-dimensional array storing the compensation amount of all points in each axis. All compensation points are based on one measure point (see Fig. 2.7.7). The user must set *dwInterval, wHome_No* and forward and backward compensation table (*nForwardTable* and *nBackwardTable*) and call the compensation setting command MCC_SetCompParam() and MCC_UpdateCompParam() to run the compensation function. The MCCL provides 256 compensation points for each axis. Each axis of the platform can be divided into a maximum of 255 compensation segments; linear compensation will be used between each segment.

When using the compensation function, the content of compensation parameter must cover the entire work course of the platform to avoid abnormal operations. Therefore, the compensation function should be enabled before the go home action has been completed. MCC_GetGoHomeStatus() can be used in combination to verify if the go home action has been completed (the return value 1 means that the go home action has been completed).

To stop the compensation function, set *dwInterval* in compensation parameters as 0. For example, execute the following programming code to stop Channel 0 compensation:

```
SYS_COMP_PARAM      stUserCompParam;

stUserCompParam.dwInterval = 0;

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

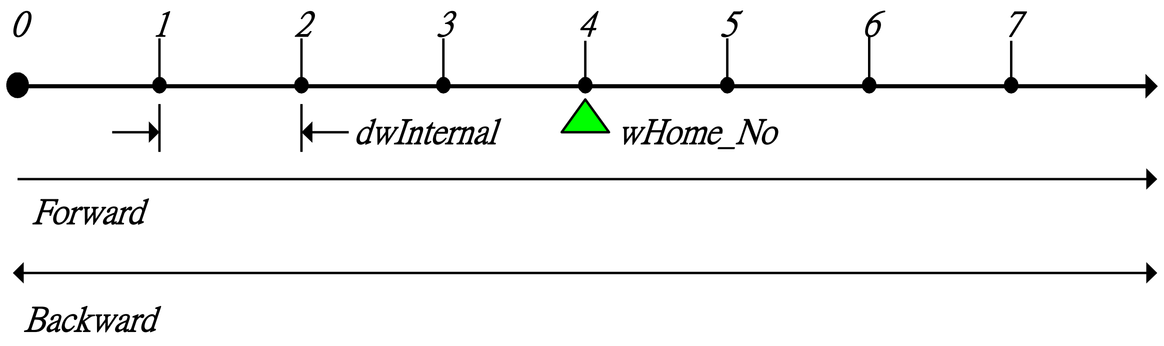


Fig. 2.7.7 Compensation segment

Compensation parameters must be set before using the compensation function. The definition of compensation parameters are as follows:

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 smaller than or equal 0, compensation function will not be performed.

wHome_No: Compensation index for the location of each axis's home position.

wPaddle: Reserved field

nForwardTable: Indicator variable for forward compensation table

nBackwardTable: Indicator variable for backward compensation table

Take Fig. 2.7.7 as an example: If the working area of X-axis is divided into 7 compensation segments, there are a total of 8 compensation points need to be measured (0 to 7). Home is located in compensation point 4 and this means that the system will believe it is currently at compensation point 4 after go home is completed. If *dwInterval* is set as 10000 (pulses), the forward work range is 10000 × (7 – 4) = 30000 (pulses) and the backward work range is 10000 × (4 - 0) = 40000 (pulses). Mechanism 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; //   unit:   pulse
stUserCompParam.nForwardTable[1]     = 20;
stUserCompParam.nForwardTable[2]     = 15;
stUserCompParam.nForwardTable[3]     = 11;
stUserCompParam.nForwardTable[4]     = 0; // home position, set as 0
stUserCompParam.nForwardTable[5]     = 10;
stUserCompParam.nForwardTable[6]     = 12;
stUserCompParam.nForwardTable[7]     = 15;

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

As explained previously, the user can divide the platform into a maximum of 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 forward 15000 pulses, from the backlash error compensation table (see stUserCompParam), it is known that the position is between the segment defined by nForwardTable[5] and nForwardTable[6] (because the position is between 10000 pulses 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 actually sends a total of 15000 + 10 + (int)((15000 – 10000)/ 10000 × 2) = 15000 + 10 + 1 = 15011 pulses.

2.8 Go Home

Users can set the go home execution order for each axis, please refer to the description in section **2.8.2-"Enabling Go Home**".

The settings of go home parameters included acceleration time, deceleration time, speed, go home direction and mode. The content of go home parameter is detailed as follows. For the meaning of each parameter, please refer to the description 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

wMode in go home parameters is used to designate the mode used in go home motion. The modes that require checking home sensor signals will confirm if the starting point is at the correct position first before conducting go home motion. In the following two conditions, it is defined that starting points are not at correct positions (assume the initial direction of go home motion is to the right):

a. Go Home motion starting point is in home sensor area (see Case 2 in Fig. 2.8.1)

b. According to the indicated motion direction, it is impossible to enter home sensor area and will trigger a limit switch (see Case 3 in Fig. 2.8.1)

If the above two situations occur, 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 when the limit switch is triggered.

b. Move at the speed set in *dfHighSpeed* to the reverse direction until entering home sensor area and continue moving. Decelerate to stop until exiting home sensor area.

c. Begin to conduct the true go home motion (the action of *Case 1*)

For various go home modes introduced in subsequent sections of this manual, Case 2 and Case 3 may occur if these modes are required to be combined with detecting home sensor signals; therefore, Case 2 and Case 3 will not be further explained. We will only explain the general situation Case 1.

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

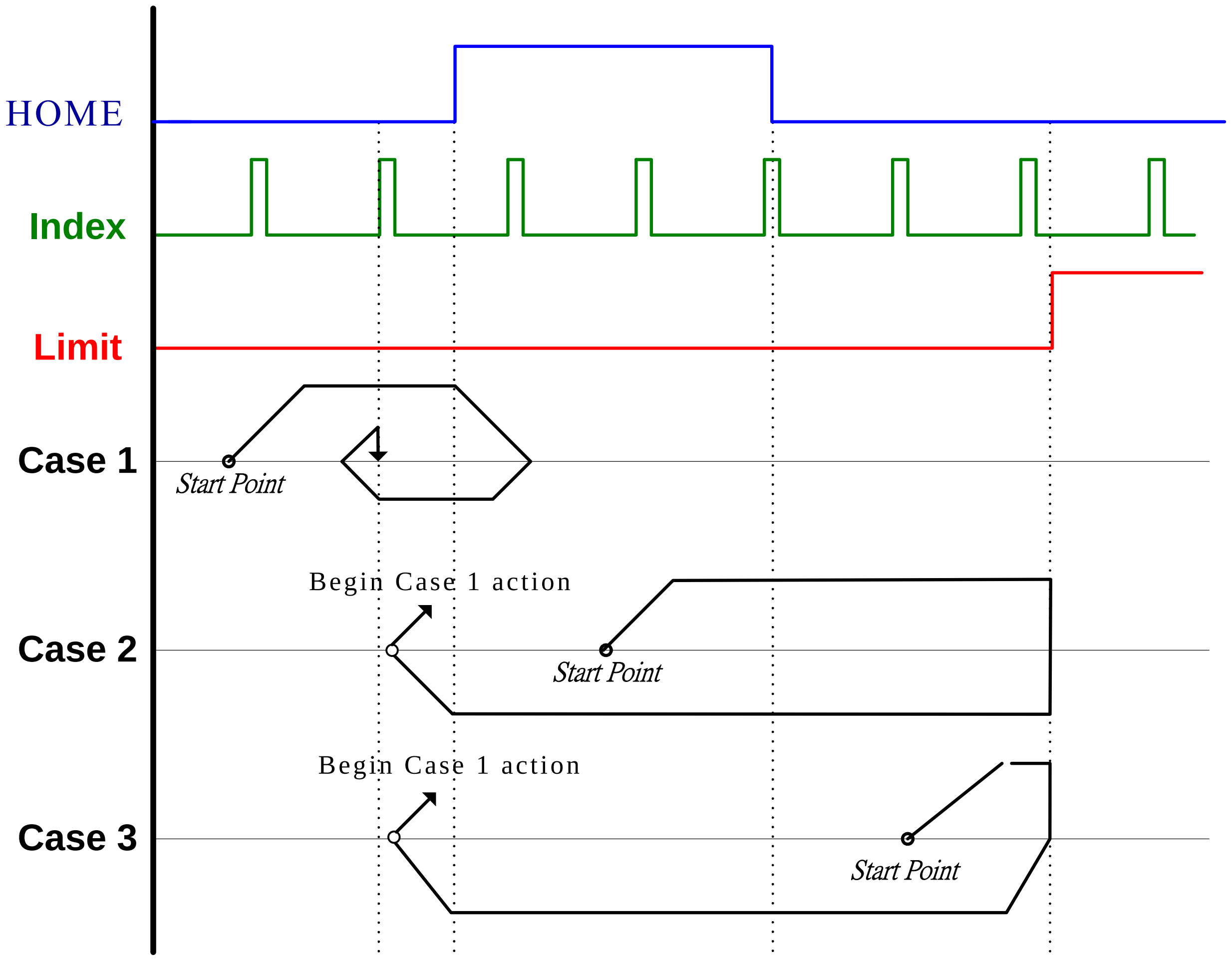
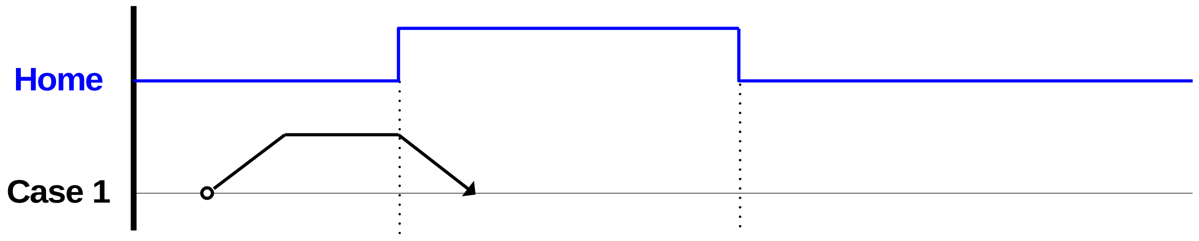


Fig. 2.8.1 Effect of different starting points of go home motion

Operational characteristics of each mode are explained as follows:

a. **Mode 3 (*wMode* = 3)** (The following description is only for Case 1; for Case 2 and Case 3, please refer to the previous description.)
Move at the speed set in *dfHighSpeed* in the indicated direction, decelerate to a stop when entering home sensor region and complete the action. (At this point, the platform will stop at themachine home position and the MCCL will move the platform to the working home position based on the parameter *dfOffset* (for details, refer to 2.4.1 & 2.4.3) and thereby complete all actions. This rule will apply to all subsequent modes).

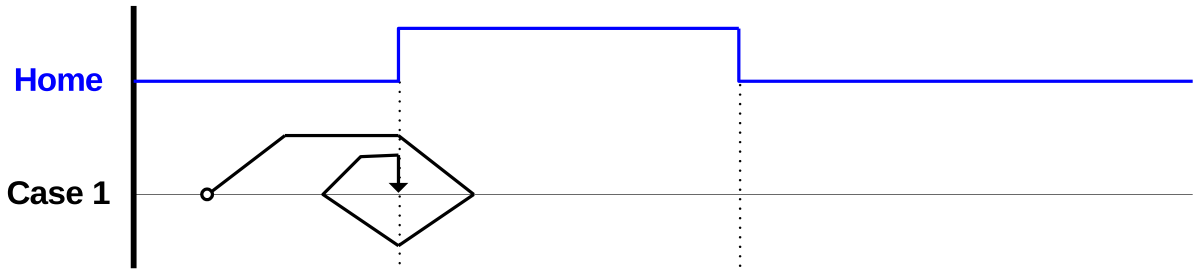


b. Mode 4 (*wMode* = 4) (The following description is only for Case 1; for Case 2 and Case 3, please refer to the previous description.)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction and decelerate to a stop when entering the home sensor region.

Step 2: Move at the speed set in *dfHighSpeed* in the reverse direction and decelerate to a stop after exiting the home sensor region.

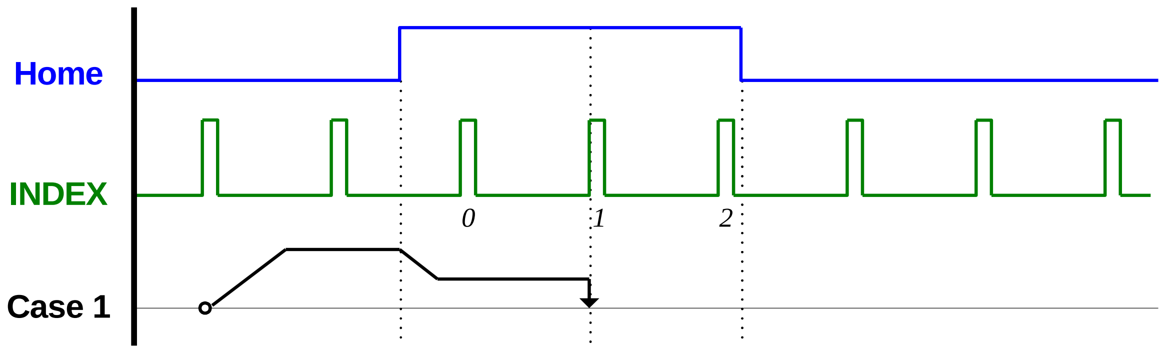
Step 3: Move at the speed set in *dfLowSpeed* in the indicated direction, execute an emergency stop after entering the home sensor region and thereby complete the action.



c. Mode 5 (*wMode* = 5) (The following description is only for Case 1; for Case 2 and Case 3, please refer to the previous description.)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction and start decelerating to *dfLowSpeed* after entering the home sensor region while simultaneously searching for the indicated index number (the example figure is set to search for index number 1, i.e. *nIndexCount* = 1).

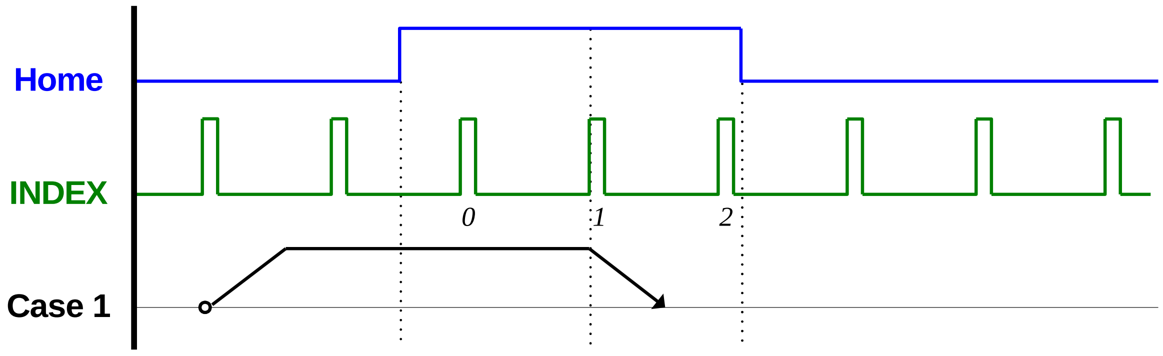
Step 2: Stop promptly after the indicated index signal is triggered and complete the action.



d. Mode 6 (*wMode* = 6) (The following description is only for Case 1; for Case 2 and Case 3, please refer to the previous description.)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction and start searching for the indicated index number after entering the home sensor region (the example figure is set to search for index number 1, i.e. *nIndexCount* = 1).

Step 2: Decelerate to a stop after the indicated index signal is triggered and complete the action.

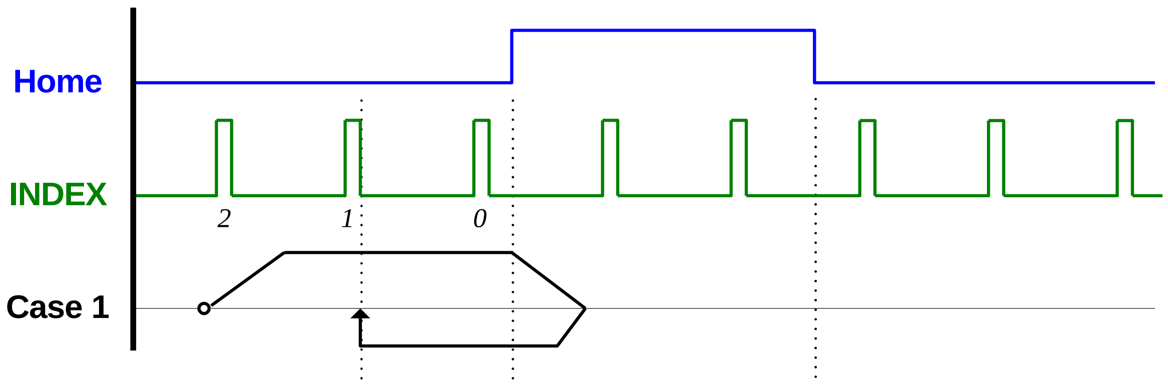


e. Mode 7 (*wMode* = 7) (The following description is only for Case 1; for Case 2 and Case 3, please refer to the previous description.)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction and decelerate to a stop when entering the home sensor region.

Step 2: Move at the speed set in *dfLowSpeed* in the reverse direction and start searching for the indicated index number after exiting the home sensor region (the example figure is set to search for index number 1, i.e. *nIndexCount* = 1).

Step 3: Stop promptly after the indicated index signal is triggered and complete the action.

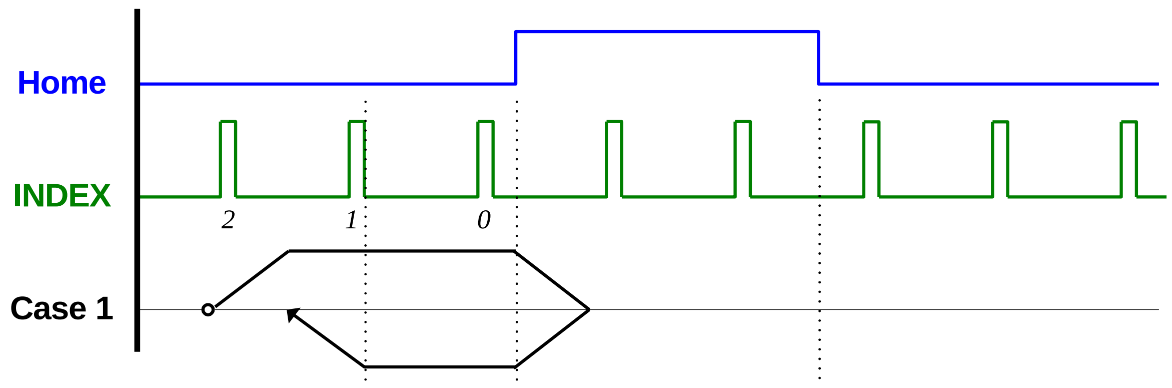


f. Mode 8 (*wMode* = 8) (The following description is only for Case 1; for Case 2 and Case 3, please refer to the previous description.)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction and decelerate to a stop when entering the home sensor region.

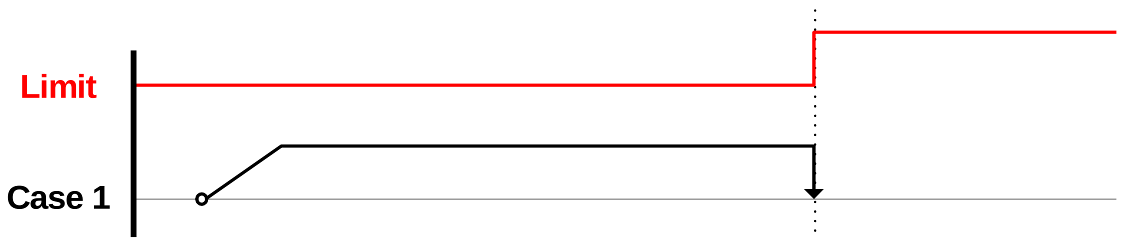
Step 2: Move at the speed set in *dfHighSpeed* in the reverse direction and start searching for the indicated index number after exiting the home sensor region (the example figure is set to search for index number 1, i.e. *nIndexCount* = 1).

Step 3: Decelerate to a stop after the indicated index signal is triggered and complete the action.



g. Mode 9 (*wMode* = 9) (There is no Case 2 or Case 3 in this mode)

Move at the speed set in *dfHighSpeed* in the indicated direction until an emergency stop is executed when the limit switch is triggered and complete the action.

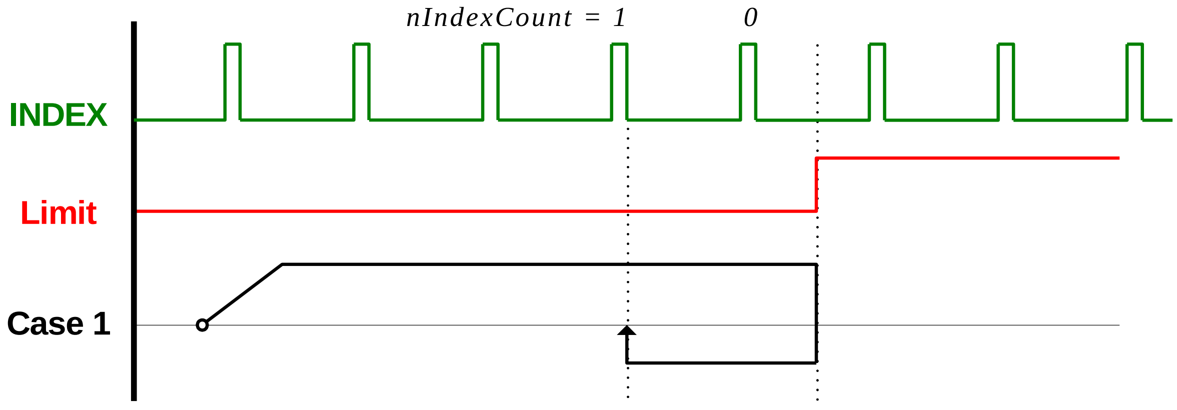


h. Mode 10 (*wMode* = 10) (There is no Case 2 or Case 3 in this mode)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction until an emergency stop is executed when the limit switch is triggered.

Step 2: Move at the speed set in *dfLowSpeed* in the reverse direction and start searching for the indicated index number (the example figure is set to search for index number 1, i.e. *nIndexCount* = 1).

Step 3: Stop promptly after the indicated index signal is triggered and complete the action.

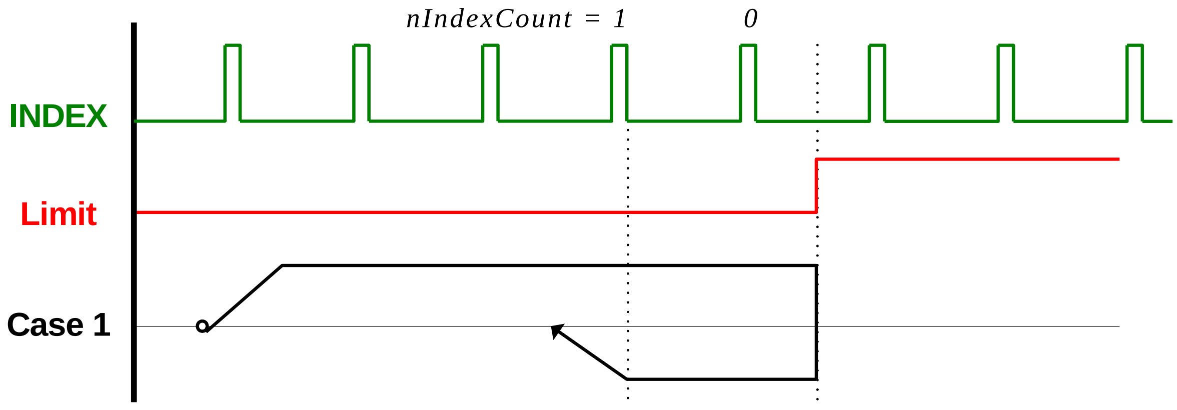


i. **Mode 11 (*wMode* = 11)** (There is no Case 2 or Case 3 in this mode)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction until an emergency stop is executed when the limit switch is triggered.

Step 2: Move at the speed set in *dfHighSpeed* in the reverse direction and start searching for the indicated index number (the example figure is set to search for index number 1, i.e. *nIndexCount* = 1).

Step 3: Decelerate to a stop after the indicated index signal is triggered and complete the action.

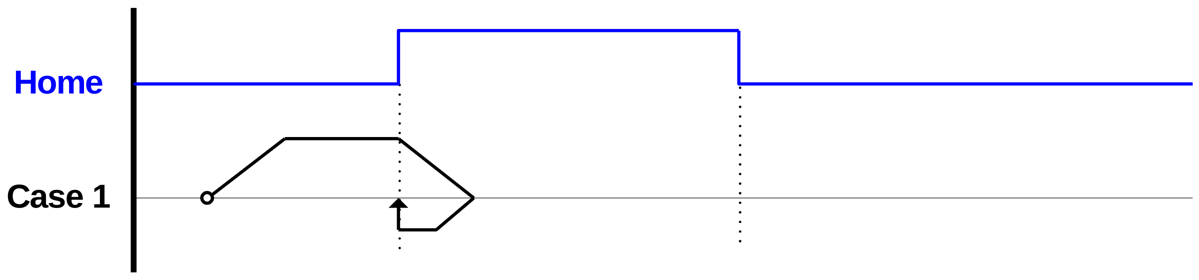


j. **Mode 12 (*wMode* = 12)** (The following description is only for Case 1; for Case 2 and Case 3, please refer to the previous description.)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction and decelerate to a stop when entering the home sensor region.

Step 2: Move at the speed set in *dfLowSpeed* in the reverse direction to exit the home sensor region.

Step 3: Stop promptly after leaving the home sensor region and complete the action.

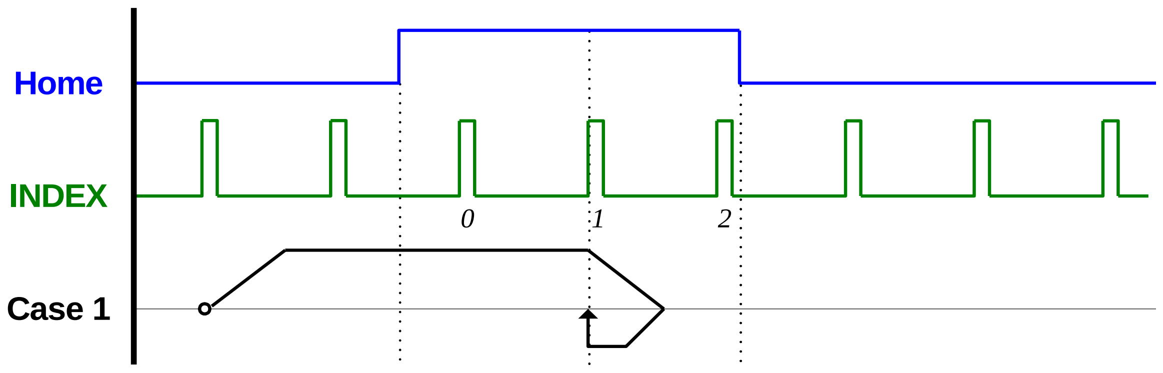


k. Mode 13 (*wMode* = 13) (The following description is only for Case 1; for Case 2 and Case 3, please refer to the previous description.)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction and start searching for the indicated index number after entering the home sensor region (the example figure is set to search for index number 1, i.e. *nIndexCount* = 1).

Step 2: After the indicated index signal is triggered, it will decelerate to a stop.

Step 3: Move at the speed set *in dfLowSpeed* in the reverse direction to return to the position where the index signal is triggered and complete the action.

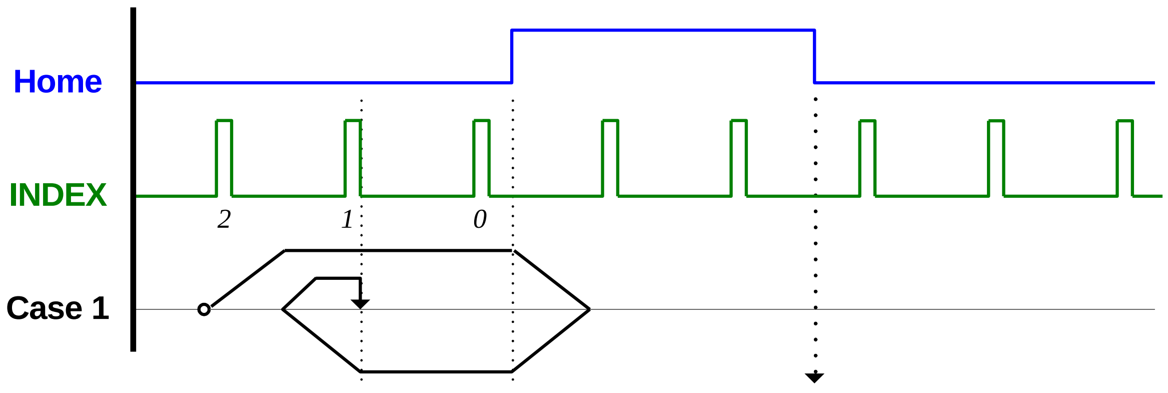


l. **Mode 14 (*wMode* = 14)** (The following description is only for Case 1; for Case 2 and Case 3, please refer to the previous description.)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction and decelerate to a stop when entering the home sensor region.

Step 2: Move at the speed set in *dfHighSpeed* in the reverse direction and start searching for the indicated index number after exiting the home sensor region (the example figure is set to search for index number 1, i.e. *nIndexCount* = 1).

Step 3: After the indicated index signal is triggered, it will decelerate to a stop.Step 4: Move at the speed set *in dfLowSpeed* in the reverse direction to return to the position where the index signal was triggered and complete the action.



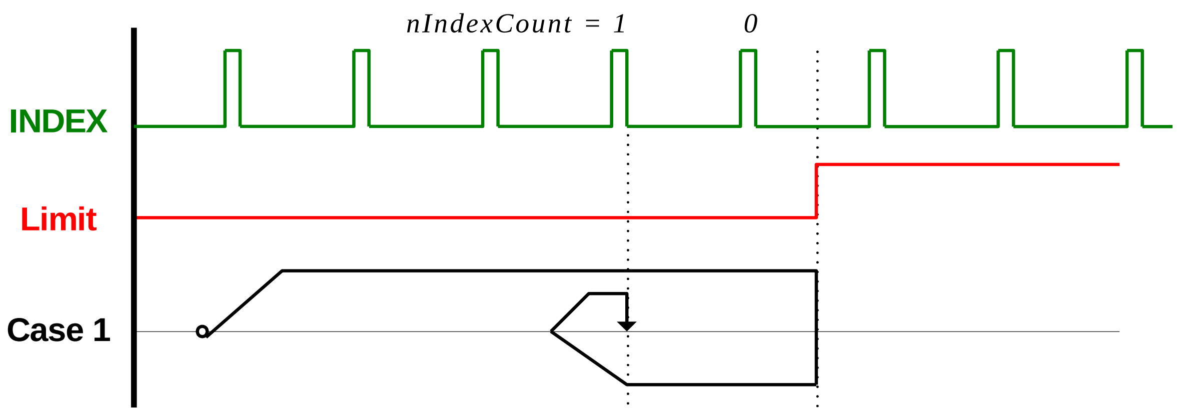
m. **Mode 15 (*wMode* = 15)** (There is no Case 2 or Case 3 in this mode)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction until an emergency stop is executed when the limit switch is triggered.

Step 2: Move at the speed set in *dfHighSpeed* in the reverse direction and start searching for the indicated index number (the example figure is set to search for index number 1, i.e. *nIndexCount* = 1).

Step 3: After the indicated index signal is triggered, it will decelerate to a stop.

Step 4: Move at the speed set *in dfLowSpeed* in the reverse direction to return to the position where the index signal was triggered and complete the action.

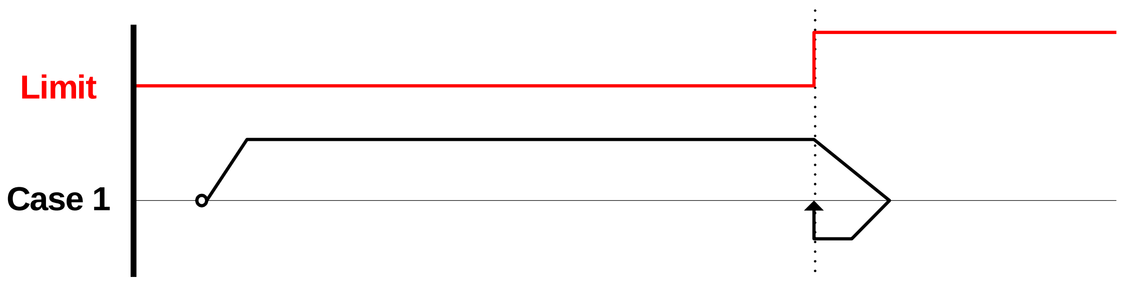


n. Mode 16 (*wMode* = 16) (There is no Case 2 or Case 3 in this mode)

Step 1: Move at the speed set in *dfHighSpeed* in the indicated direction and decelerate to a stop when the limit switch is triggered.

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

Step 3: Stop promptly when leaving the limit switch region and complete the action.



2.8.2 Enabling Go Home

The procedure for enabling go home is detailed as follows.

1. First, use MCC_SetHomeConfig() to set the go home parameters (please refer to the description in previous sections)

2. Call MCC_Home(

```
int nOrder0,   int nOrder1,   int nOrder2,
int nOrder3,   int nOrder4,   int nOrder5, ,
int nOrder6,   int nOrder7,   WORD   wCardIndex)
```

where

nOrder0 ~ nOrder7	Go home execution order for each axis
wCardIndex	motion control card index

The go home execution order for each axis can be set from 0 to 7, 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. When these actions are completed, it will execute the go home execution for motion axes set as 1 and follow this principle to finish the go home action of all motion axes. The order set as 255 indicates that the go home action will not be performed on 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 identify if the go home action has been completed. If the return value is 1, it indicates that the go home action has been completed and the value of 0 indicates that the action is still under processing.

CAUTION

1. The go home process can be divided into 3 phases no matter which mode is implemented:
   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 position to the working home position
2. When there are multiple axes performing go home at the same time, all axes must complete Phase 1 before entering Phase 2 together. Similarly, all axes must complete Phase 2 before entering Phase 3 together. Therefore, it is possible and normal that during the go home process, an axis that has completed the given phase has to stop moving and wait until other axes

complete the same phase.

The following table lists the phases included in each go home mode:

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

2.9 Local I/O Control

Local I/O refers to the input and output connections built in the IMP and is different from the Asynchronous Remote I/O Board (IMP-ARIO) that can expand to a maximum of 512 input and 512 output connections (IMP-ARIO is optional). Each of these I/O connections has its specific purpose; however, if these specific purposes are not required in the actual application (for example, when the limit switch check function or output servo-on/off signal is not necessary), these I/O connections can be used as general I/O connections as well.

2.9.1 Input Connection Status

The built-in input connections on the IMP include:

a. 8 home sensor signal input connections. MCC_GetHomeSensorStatus() can be used to acquire the home sensor input signal status.

b. 8 positive limit switch signal input connections and 8 negative limit switch signal input connections. MCC_GetLimitSwitchStatus() can be used to acquire the limit switch input signal status.

c. 1 emergency stop switch signal input connection. MCC_GetEmgcStopStatus() can be used to acquire its input signal status.

2.9.2 Signal Output Control

The built-in output connections on the IMP include:

a. 8 servo-on/off signal control connections. MCC_SetServoOn() and MCC_SetServoOff() can be used to output the servo-on/off signal.

b. 1 position ready signal control connection. MCC_EnablePosReady() and

MCC_DisablePosReady() can be used to output or cancel the position ready signal. To ensure the safety of the operator, after using MCC_InitSystem() to successfully activate the system and the system is verified to be under normal operation, it is necessary to use the position ready signal additionally to activate external circuits (such as the driver or motor circuits).

c. 8 LED indicators which can be used to display the output indicator.

2.9.3 Input Signal Triggered Interrupt Service Routine

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

a. a total of 24 inputs of the IMP-2, including:

Channel 0 Limit Switch +(OTP0)
Channel 1 Limit Switch +(OTP1)
Channel 2 Limit Switch +(OTP2)
Channel 3 Limit Switch +(OTP3)
Channel 4 Limit Switch +(OTP4)
Channel 5 Limit Switch +(OTP5)
Channel 6 Limit Switch +(OTP6)
Channel 7 Limit Switch +(OTP7)
Channel 0 Limit Switch -(OTN0)
Channel 1 Limit Switch -(OTN1)
Channel 2 Limit Switch -(OTN2)
Channel 3 Limit Switch -(OTN3)
Channel 4 Limit Switch -(OTN4)
Channel 5 Limit Switch -(OTN5)
Channel 6 Limit Switch -(OTN6)
Channel 7 Limit Switch -(OTN7)
Channel 0 Home Switch (HOME0)

Channel 1 Home Switch (HOME1)

Channel 2 Home Switch (HOME2)

Channel 3 Home Switch (HOME3)

Channel 4 Home Switch (HOME4)

Channel 5 Home Switch (HOME5)

Channel 6 Home Switch (HOME6)

Channel 7 Home Switch (HOME7)

The procedure of using "input connection triggered interrupt service routine" is detailed below:

Step 1: **Use MCC_SetLIORoutineEx() 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 ****LIOISR*)(***LIOINT****)**

For example, the customized command can be designed as follows:

```
stdcall MyLIOFunction(LIOINT *pstINTSource)
{
    //  determine whether the routine was called because Channel 0 Limit Switch +
was triggered
    if (pstINTSource->OTP0)
    {
        //  handling procedure when Channel 0 Limit Switch + is triggered
    }

    //  determine whether the routine was called because Channel 1 Limit Switch +
was triggered
    if (pstINTSource->OTP1)
    {
        //    handling procedure when Channel 1 Limit Switch + is triggered
    }
}
```

A routine such as "else if (pstINTSource->*OTP1*)" cannot be used, because pstINTSource->*OTP0* and pstINTSource->*OTP1* may not be 0 simultaneously.

Later, use MCC_SetLIORoutineEx (MyLIOFunction) to serially connect the customized ISR. When the customized is triggered during execution, the system can use the pstINTSource parameter declared as *LIOINT* imported into the customized routine to determine which input connection is triggered when the customized routine was called. The definition of *LIOINT* is as follows:

```
typedef struct _LIO_INT
{
   BYTE OTP0;
   BYTE OTP1;
   BYTE OTP2;
   BYTE OTP3;
   BYTE OTP4;
   BYTE OTP5;
   BYTE OTP6;
   BYTE OTP7;
   BYTE OTN0;
   BYTE OTN1;
   BYTE OTN2;
   BYTE OTN3;
   BYTE OTN4;
   BYTE OTN5;
   BYTE OTN6;
   BYTE OTN7;
   BYTE HOME0;
   BYTE HOME1;
   BYTE HOME2;
   BYTE HOME3;
   BYTE HOME4;
   BYTE HOME5;
   BYTE HOME6;
   BYTE HOME7;
} LIOINT;
```

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

	IMP-2
OTP0	Channel 0 Limit Switch+
OTP1	Channel 1 Limit Switch+
OTP2	Channel 2 Limit Switch+
LDI3	Channel 3 Limit Switch+
LDI4	Channel 4 Limit Switch+
OTP5	Channel 5 Limit Switch+
OTP6	Channel 6 Limit Switch+
OTP7	Channel 7 Limit Switch+
OTN0	Channel 0 Limit Switch-
OTN1	Channel 1 Limit Switch-
OTN2	Channel 2 Limit Switch-
OTN3	Channel 3 Limit Switch-
OTN4	Channel 4 Limit Switch-
OTN5	Channel 5 Limit Switch-
OTN6	Channel 6 Limit Switch-
OTN7	Channel 7 Limit Switch-
HOME0	Channel 0 HOME Switch
HOME1	Channel 1 HOME Switch
HOME2	Channel 2 HOME Switch
HOME3	Channel 3 HOME Switch
HOME4	Channel 4 HOME Switch
HOME5	Channel 5 HOME Switch
HOME6	Channel 6 HOME Switch
HOME7	Channel 7 HOME Switch

If the value of these fields does not equal 0, then currently the connection corresponding to that field has a signal input. For example, if the MyLIOFunction() input parameter pstINTSource->*OTP2* is not 0, it contacts with 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. The input parameters of MCC_SetLIOTriggerType() can be:

LIO_INT_NO	Not triggered
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 Signal Triggered Interrupt Service Routine" function. MCC_DisableLIOTrigger() can be used to disable this function.**

2.10 Encoder Control

The encoder control functions provided by the MCCL include counts per encoder cycle selection, count acquisition, count latch, index trigger, and automatic count comparison and trigger.

Before using the encoder control functions, the fields related to encoder characteristics in mechanism parameters must be set correctly. Please refer to the description in section **2.4.2-“Encoder Parameters**” for details.

2.10.1 General Control

If the encoder parameter (see section 2.4.2) *wType* is set as ENC_TYPE_AB, meaning the input form is set as A/B Phase, then MCC_SetENCInputRate() can be used to set the counts per encoder cycle.

The counts per encoder cycle can be ×1, ×2 or ×4, representing 1, 2 or 4 counts per encoder cycle respectively.

The counts per encoder cycle can be 1, 2, or 4, representing the counts per encoder cycle of×1, ×2 or ×4 respectively.

If the mechanism parameter wCommandMode is set to OCM_VOLTAGE (using V Command) and the counts per encoder cycle has been changed, then it is necessary to reset the mechanism parameter *dwPPR*. MCC_GetENCValue() can be used to obtain the encoder count.

2.10.2 Count Latch

The MCCL provides a “count latch” function that allows users to set the signal sources triggering the encoder count to be recorded in the latched register. MCC_GetENCLatchValue() can be used to acquire the record in the latched register. The procedure of using “count latch” is detailed below:

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

All of the following trigger sources can trigger the encoder count to be recorded in the latched register. MCC_SetENCLatchSource() is used to set the trigger source. Multiple criteria can be obtained during setting. These trigger signal sources include:

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
ENC_TRIG_INDEX5	Index signal in encoder Channel 5
ENC_TRIG_INDEX6	Index signal in encoder Channel 6
ENC_TRIG_INDEX7	Index signal in encoder Channel 7
ENC_TRIG_OTP0	Interrupt request from local input connection OT0+
ENC_TRIG_OTP1	Interrupt request from local input connection OT1+
ENC_TRIG_OTP2	Interrupt request from local input connection OT2+
ENC_TRIG_OTP3	Interrupt request from local input connection OT3+
ENC_TRIG_OTP4	Interrupt request from local input connection OT4+
ENC_TRIG_OTP5	Interrupt request from local input connection OT5+
ENC_TRIG_OTP6	Interrupt request from local input connection OT6+
ENC_TRIG_OTP7	Interrupt request from local input connection OT7+
ENC_TRIG_OTN0	Interrupt request from local input connection OT0-
ENC_TRIG_OTN1	Interrupt request from local input connection OT1-
ENC_TRIG_OTN2	Interrupt request from local input connection OT2-
ENC_TRIG_OTN3	Interrupt request from local input connection OT3-
ENC_TRIG_OTN4	Interrupt request from local input connection OT4-
ENC_TRIG_OTN5	Interrupt request from local input connection OT5-
ENC_TRIG_OTN6	Interrupt request from local input connection OT6-
ENC_TRIG_OTN7	Interrupt request from local input connection OT7-

Using

MCC_SetENCTriggerSource(ENC_TRIG_INDEX0 | ENC_TRIG_OTP0, 0, 0)

means that when the encoder Channel 0 index is input or the positive direction limit of Channel 0 is triggered, the encoder count will be recorded in the latched register of 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 latch count mode. The selectable modes include:

ENC_TRIG_FIRST The count is immediately latched and changes no more when the first trigger criterion is met.

ENC_TRIG_LAST The latched count is updated unlimited times when the trigger criteria are met.

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

The MCCL is not equipped with the command that can be used to identify if the record in the latched register has been updated. However, all trigger sources that can update the latched register record are capable of triggering the ISR as well. Users can use this function to identify that the record has been updated and use MCC_GetENCLatchValue() to obtain the updated record. For more details, please refer to "**IMP 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 can set the comparative value for encoder channels 0 to 7; after the function has been enabled for the selected channel, the user-customizable ISR will be called automatically when the given channel count equals the set comparative value. The procedure of using "encoder count triggered interrupt service routine" is detailed below:

***Step 1*: Use MCC_SetENCRoutine() to serially connect the customized ISR**

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

typedef void(_stdcall **ENCISR*)(*ENCINT)**

For example, the customized command can be designed as follows:

```
stdcall MyENCFunction(ENCINT *pstINTSource)
{

    //  determine whether the routine was triggered because the count of encoder
Channel 0 equals the comparative value
    if (pstINTSource->COMP0)
    {
     //  handling procedure when the comparative value conditions of Channel 0 are
met
     }

    //  determine whether the routine was triggered because the count of encoder
Channel 1 equals the comparative value
    if (pstINTSource->COMP1)
    {
        //  handling procedure when the comparative value conditions of Channel 1
    are met
    }

    //  determine whether the routine was triggered because the count of encoder
Channel 2 equals the comparative value
    if (pstINTSource->COMP2)
    {
        //  handling procedure when the comparative value conditions of Channel 2
    are met
    }

    //  determine whether the routine was triggered because the count of encoder
Channel 3 equals the comparative value
    if (pstINTSource->COMP3)
    {
        //  handling procedure when the comparative value conditions of Channel 3
    are met
    }

    //  determine whether the routine was triggered because the count of encoder
Channel 4 equals the comparative value
    if (pstINTSource->COMP4)
    {
        //  handling procedure when the comparative value conditions of Channel 4
    are met
    }

    //  determine whether the routine was triggered because the count of encoder
Channel 5 equals the comparative value
    if (pstINTSource->COMP5)
    {
        //  handling procedure when the comparative value conditions of Channel 5
    are met
    }
    //  determine whether the routine was triggered because the count of encoder
Channel 6 equals the comparative value
    if (pstINTSource->COMP6)
    {
        //  handling procedure when the comparative value conditions of Channel 6
    are met
    }
    //   determine whether the routine was triggered because the count of encoder
Channel 7 equals the comparative value
    if (pstINTSource->COMP7)
    {
        //   handling procedure when the comparative value conditions of Channel 7
    are met
    }

}
```

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

Later, use MCC_SetENCRoutine (MyENCFunction) to serially connect to the customized ISR. When the ISR is triggered during execution, the system can use the pstINTSource parameter, declared as *ENCINT*,*to determine which trigger criterion was satisfied when the routine was called. The definition of *ENCINT* is as follows:

```
typedef struct _ENC_INT
{
     BYTE      COMP0;
     BYTE      COMP1;
     BYTE      COMP2;
     BYTE      COMP3;
     BYTE      COMP4;
     BYTE      COMP5;
     BYTE      COMP6;
     BYTE      COMP7;
     BYTE      INDEX0;
     BYTE      INDEX1;
     BYTE      INDEX2;
     BYTE      INDEX3;
     BYTE      INDEX4;
```

```
        BYTE        INDEX5;
        BYTE        INDEX6;
        BYTE        INDEX7;
} ENCINT;
```

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

COMP0	The count of encoder Channel 0 equals the set comparative value
COMP1	The count of encoder Channel 1 equals the set comparative value
COMP2	The count of encoder Channel 2 equals the set comparative value
COMP3	The count of encoder Channel 3 equals the set comparative value
COMP4	The count of encoder Channel 4 equals the set comparative value
COMP5	The count of encoder Channel 5 equals the set comparative value
COMP6	The count of encoder Channel 6 equals the set comparative value
COMP7	The count of encoder Channel 7 equals the set comparative value
INDEX0	Triggered by the index signal of encoder Channel 0
INDEX1	Triggered by the index signal of encoder Channel 1
INDEX2	Triggered by the index signal of encoder Channel 2
INDEX3	Triggered by the index signal of encoder Channel 3
INDEX4	Triggered by the index signal of encoder Channel 4
INDEX5	Triggered by the index signal of encoder Channel 5
INDEX6	Triggered by the index signal of encoder Channel 6
INDEX7	Triggered by the index signal of encoder Channel 7

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

***Step 3*: Use MCC_EnableENCCompTrigger() to enable the “ encoder count triggered interrupt service routine” function for the indicated channel. 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 can trigger the ISR by using index signals of encoder channels 0 to 7. The procedure for using "encoder index triggered interrupt service routine" function is detailed below:

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

If MCC_SetENCRoutine() has not been called before, please refer to the previous description of this step (Section 2.10.3, ***Step 1***); if MCC_SetENCRoutine() has been called before, simply determine which channel triggered the function using the "index signal input" fields (*INDEX0 ~ INDEX7*) of pstINTSource in the customized routine. Please refer to the following example:

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

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

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

```
        }

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

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

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

        //    determine whether the routine was triggered by the index signal of Channel 6
        if (pstINTSource->INDEX6)
        {
                //    handling procedure when the index signal of encoder Channel 6 is input
        }

        //    determine whether the routine was triggered by the index signal of Channel 7
        if (pstINTSource->INDEX7)
        {
                //    handling procedure when the index signal of encoder Channel 7 is input
        }

}
```

Step 2**:** **Use MCC_EnableENCIndexTrigger() to activate encoder index triggered function for the indicated channel. MCC_DisableENCIndexTrigger() can be used to disable this function.**

This function can work with "encoder count latch" to obtain the count when the index signal is input. (For "encoder count latch" function, please refer to the description in the previous section). MCC_GetENCIndexStatus() can be used to identify whether the current motor position is on the encoder index position.

2.11 Analog Voltage Output (D/A Converter，DAC) Control

If motion axes that are required to output voltage have been programmed in mechanism parameters as V Command motion axes (i.e. *nCommandMode* is set as OCM_VOLTAGE), it is not possible to use any of the following commands related to DAC. An incorrect return value will be obtained by calling these commands. Users are advised to pay extra attention to this.

2.11.1 General Control

After activating with the MCCL by MCC_InitSystem(), MCC_SetDACOutput() can be used to output analog voltage. Voltage output range is -10V to +10V.

Meanwhile, MCC_StopDACConv() can be used to disable analog voltage output function and MCC_StartDACConv() can be used to restart this function.

2.11.2 Output Voltage Hardware Trigger Mode

The "output voltage hardware trigger mode" provided in the MCCL can program one output voltage value for the selected DAC channel in advance and can trigger this preset voltage by a specific hardware trigger source. This function will be directly handled by the hardware after programming to ensure the best instantaneousness. The procedure of using "output voltage hardware trigger mode" is detailed below:

***Step 1*: Use MCC_SetDACTriggerOutput() to program the output voltage value in advance**

For example, use MCC_SetDACTriggerOutput (2.0, 1, 0) to program DAC Channel 1 of Card 0 to output 2.0V in advance.

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

Possible hardware trigger sources are defined below; it is also possible to set multiple trigger conditions at the same time. Please note that all these hardware trigger sources are required to be from the same motion control card.

1.	DAC_TRIG_ENC0	Specific count in encoder Channel 0
2.	DAC_TRIG_ENC1	Specific count in encoder Channel 1
3.	DAC_TRIG_ENC2	Specific count in encoder Channel 2
4.	DAC_TRIG_ENC3	Specific count in encoder Channel 3
5.	DAC_TRIG_ENC4	Specific count in encoder Channel 4
6.	DAC_TRIG_ENC5	Specific count in encoder Channel 5
7.	DAC_TRIG_ENC6	Specific count in encoder Channel 6
8.	DAC_TRIG_ENC7	Specific count in encoder Channel 7

ISRs related to these hardware trigger sources should also be enabled when setting the hardware trigger source so that these sources can trigger the output voltage. For example, when using MCC_SetDACTriggerSource(*DAC_TRIG_ENC0*, 1, 2) to set the specific count of encoder Channel 0 in Card 2 as the DAC hardware trigger source of Channel 1 in Card 2, it is required to enable the “encoder count triggered interrupt service routine” function of Channel 0, which means to use MCC_SetENCCompValue() and MCC_EnableENCCompTrigger() to enable the encoder ISR function of Channel 0. For details of this function, please refer to section **2.10.3-”Encoder Count Triggered Interrupt Service Routine”**. Similarly, when setting the hardware trigger source to be the signal of limit switch, it is also required to use MCC_SetLIOTriggerType() and MCC_EnableLIOTrigger() to activate input signal triggered interrupt service routine. For details of this function, please refer to the section **2.9.3-”Input Signal Triggered Service Routine”**.

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

2.12 Analog voltage input (A/D Converter, ADC) Control

2.12.1 Initial Settings

For IMPs, it is necessary to complete following steps before using the "analog voltage input control" function.

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

(1) MCC_SetADCConvType (*ADC_TYPE_BIP*): use bipolar converter type and -5V to 5V readable voltage range.

(2) MCC_SetADCConvType (*ADC_TYPE_UNI*) :use unipolar converter type and 0V to 10V readable voltage range.

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

(1) MCC_SetADCConvMode (*ADC_MODE_FREE*):conduct continuous voltage acquisition. The voltage acquired will vary based on different input voltages. This command is required to be used in combination with MCC_EnableADCConvChannel(). For details of this function, please refer to section **2.12.2- "Continuous Voltage Conversion"**.

(2) MCC_SetADCConvMode (*ADC_MODE_SINGLE*): conduct single voltage acquisition; unless MCC_StartADCConv() is called again, the value acquired will not change. This command is required to be used in combination with MCC_SetADCSingleChannel(). For details of this function, please refer to section **2.12.3- "Single Channel Voltage Conversion".**

2.12.2 Continuous Voltage Conversion

After completing the initial settings mentioned previously, follow the steps below to acquire the input voltage of a specific channel:

***Step 1*: Use MCC_SetADCConvMode(*ADC_MODE_FREE*)**

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

input analog voltage

A maximum of eight A/D channels are allowed to input analog voltage simultaneously. The voltage is only converted in the permitted input channels. MCC_DisableADCConvChannel() can be used to inhibit the selected channel to input analog voltage.

***Step 3*: Use MCC_StartADCConv() to activate and MCC_StopADCConv() to disable the analog voltage input function.**

***Step 4*: Use MCC_GetADCInput() to obtain the voltage value**

2.12.3 Single Channel Voltage Conversion

The MCC_SetADCSingleChannel() provided in the MCCL can select a specific channel to be the only one allowed to convert voltage while conversion function of other channels are disabled.

Use MCC_SetADCSingleChannel() to select the channel and call MCC_SetADCConvMode (*ADC_MODE_SINGLE*) to use the single conversion mode. This selected channel will convert the voltage once after calling MCC_StartADCConv(). There will be no more conversion after it is completed. The user must call MCC_StartADCConv() again to conduct the next single conversion. During the conversion period (around 8μs), MCC_GetADCWorkStatus() can be used to confirm if the action is completed. After the conversion completion is confirmed, **MCC_GetADCInput() can be used to obtain the input voltage.**

2.12.4 Specific Voltage Triggered Interrupt Service Routine

The “specific voltage triggered interrupt service routine” function provided in the MCCL can set the voltage comparative value of ADC channel. When this function is enabled and the trigger conditions are met, the user-customizable ISR will be automatically called. The procedure of using “specific voltage triggered interrupt service routine” is detailed 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)**

For example, the customized command can be designed as follows:

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

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

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

Later, use MCC_SetADCRoutine(MyADCFunction) to serially connect the customized ISR. When the ISR is triggered during execution, the system can use the pstINTSource parameter declared as *ADCINT* in the customized routine to determine which trigger criterion was satisfied when the customized routine was called. The definition of *ADCINT* is as follows:

```
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 call, by a field value, are presented below:

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

***Step 2*: Refer to the previous description to complete "initial settings"**

***Step 3*: Use MCC_SetADCCompValue() to set the comparative value of voltage comparator**

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

When the input voltage is compared with the set comparative value, the three smallest bits can be masked by comparison to reduce the sensitivity of the comparator and prevent interrupts resulting from input voltage vibrations. The parameters that can be set by this command include:

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

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

The voltage comparison mode is used to set the conditions for triggering interrupts. The voltage comparison modes can be set as follows:

ADC_COMP_RISE The ADC input voltage passes the comparison value while increasing.

ADC_COMP_FALL The ADC input voltage passes the comparison value while decreasing.

ADC_COMP_LEVEL The ADC input voltage passes the comparison value while being changed.

***Step 6*: Use MCC_EnableADCCompTrigger() to activate this function**

***Step 7*: Use in combination with "continuous voltage conversion" or "single channel voltage conversion" function**

2.12.5 Voltage Conversion Completion Triggered Interrupt Service Routine

The two types of "voltage conversion completion triggered interrupt service routine" provided is the MCCL are detailed below:

I. The interrupt service routine is triggered after any ADC channel completes voltage conversion. The procedure of using this function is as follows:

***Step1*: Use MCC_SetADCRoutine() to serially connect the customized interrupt service routine**

If MCC_SetADCRoutine() has not been called, please refer to the previous description of this step; if MCC_SetADCRoutine() has been called, simply add the parameter pstINTSource "voltage conversion completion" field (*CONV*) determination to the customized routine. Please refer to the following example:

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

***Step2*: Use MCC_EnableADCConvTrigger() to enable and MCC_DisableADCConvTrigger() to disable this function.**

II. The interrupt service routine is triggered when ADC tagged channel completes voltage conversion. The procedure of using this function is as follows:

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

service routine

If MCC_SetADCRoutine() has not been called before, please refer to the previous description of this step (Section 2.12.4, ***Step 1***); if MCC_SetADCRoutine() has been called before, simply determine which channel triggered the function using the "tagged channel voltage conversion completion" field (*TAG*) of pstINTSource in the customized routine. Please refer to the following example:

```
_stdcall MyADCFunction(ADCINT *pstINTSource)
{
    //   determine whether the routine was triggered because the ADC tagged channel
completes voltage conversion
    if (pstINTSource->TAG)
    {
     //   handling procedure when the tagged channel completes voltage 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 Time and Watchdog Control

2.13.1 Timer Triggered Interrupt Service Routine

The length of the 32-bit timer on the IMP can be set by using the MCCL. When the timer function is activated and the timing is completed (i.e., the value of timer equals the set value), it will trigger the user-customizable ISR and restart timing. This process will continue until this function is disabled. The procedure of using "timer triggered interrupt service routine" is detailed below:

***Step 1*: Use MCC_SetTMRRoutine() 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 **TMRISR*)(*TMRINT)**

Call MCC_SetTMRRoutine() and in the customized routine use the "timing ends" field (*TIMER*) of pstINTSource to determine whether the routine was triggered. Please refer to the following example:

```
stdcall MyTMRFunction(TMRINT *pstINTSource)
{
    //   determine whether the routine was triggered because the timer ends
    if (pstINTSource->TIMER)
    {
      //   handling procedure when the timer ends
    }
}
```

***Step 2*: Use MCC_SetTimer() to set the timer in the unit of System Clock (10ns)**

***Step 3:* Use MCC_EnableTimerTrigger() to enable the "timer triggered interrupt**

service routine" function and MCC_DisableTimerTrigger() to disable the function.

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

2.13.2 Watchdog Control

After the user has enabled the watchdog function, it is necessary to use MCC_RefreshWatchDogTimer() to clear the watchdog timer before the timer finishes (i.e., before the watchdog timer value equals the set comparative value). Otherwise, once the watchdog timer value equals to the set comparative value, the hardware will be reset. The procedure of using the watchdog is as follows:

***Step 1*: Use MCC_SetTimer() to set the core timer; the timing period is a 32-bit numerical value and the unit is System Clock (10ns).**

***Step 2*: MCC_SetWatchDogTimer() set the watchdog timer period**

The watchdog timer is a 32-bit numerical value and uses the time of the core timer as the time base. In other words, if the following programming code is used:

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

At this point, it means that the comparison value of Card 0 watchdog timer is set as (10ns × 1000000)× 2000 = 20s.

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

This command can program the signal duration of the watchdog generated hardware reset (32-bit numerical value with the maximum of 4294967296; unit: system clock (10ns)).

***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" function. The user will be alerted before the watchdog resets the hardware and conduct the necessary handling within the timer ISR.

2.14 Remote I/O Control

2.14.1 Initial Settings

Each IMP consists of one Remote I/O control card plug referred to as Remote I/O Master which is capable of controlling 32 Remote I/O control cards (index IMP-ARIO; called as Remote I/O Slave). Each Remote I/O control card provides 16 output and 16 input connections as shown in the following figure:

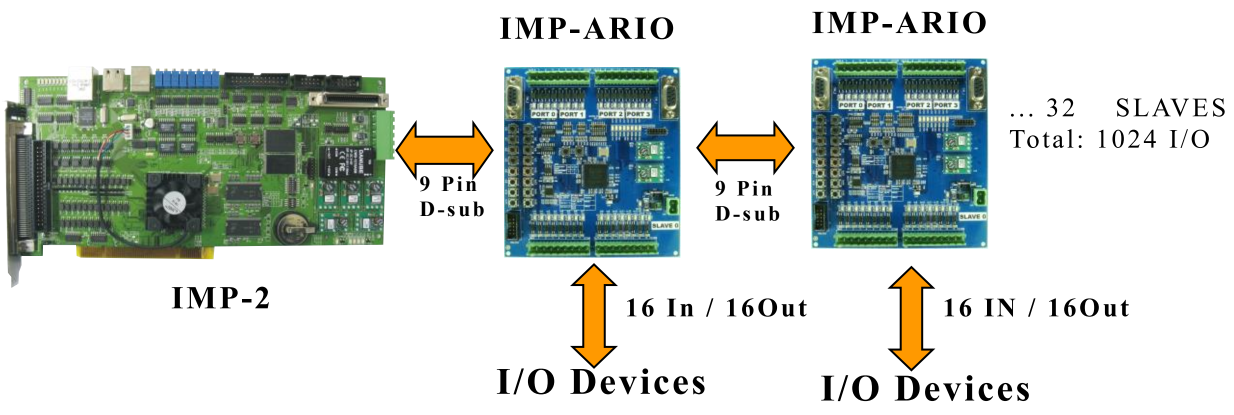


Fig. 2.14.1 Remote Master and Slave

Use MCC_EnableARIOSlaveControl() to enable the data transmission function. The example is as follows:

MCC_EnableARIOSlaveControl(WORD wSet, WORD wSlave, WORD wCardIndex=0);

2.14.2 Setting and Acquiring I/O Status

After finishing initial settings, MCC_GetARIOInputValue() can be used to acquire the input signal status and MCC_SetARIOOutputValue() can be used to set the output signal status. The prototype of MCC_GetARIOInputValue() is as follows:

```
MCC_GetARIOInputValue( WORD*   pwValue,
                       WORD    wSet,
                       WORD    wSlave,
                                        WORD    wCardIndex);
```

The Remote I/O acquisition function can acquire the statuses of 16 slave input connections. Bit 0 to bit 15 in *pwValue store the statuses of Input 0 to Input 15 separately. Parameter *wSet* currently does not have any usage *and* parameter *wSlave* is used to indicate the slave to be acquired by Remote IO control card.

Remote I/O written function can set up to 16 slave connection output statuses. Therefore, the usage of MCC_SetARIOOutputValue() is similar to MCC_GetARIOInputValue(); the statuses of 16 output connections of the indicated group must also be set whenever setting the output connection status. The prototype of MCC_SetARIOOutputValue() is as follows:

```
MCC_SetARIOOutputValue( WORD    wValue,
                        WORD    wSet,
                        WORD    wSlave,
                        WORD    wCardIndex);
```

where the statuses of 16 output connections are indicated by wValue

3. A^+ PC Mode Development Environment

3.1 Using Visual C++

Include Files

MCCL.h

MCCL_Fun.h

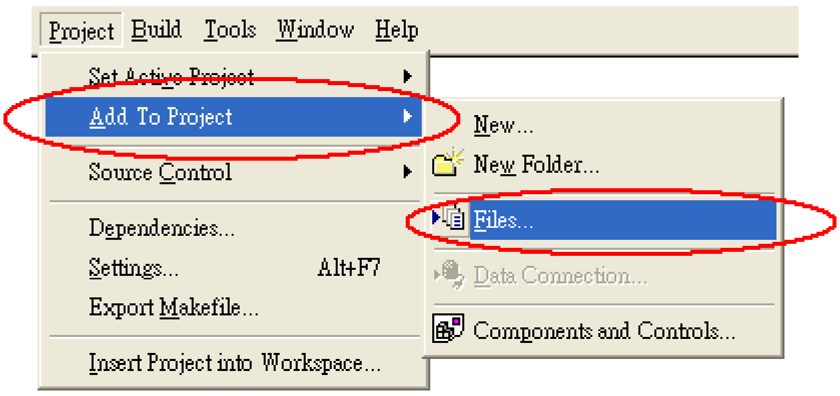
Import Library **(Users must add this file to the project)**

MCCLPCI_IMP.lib (for A^+ PC mode)

The following is the process for adding the necessary import library **MCCLPCI_IMP.lib** into the project when using an IMP and implementing VC++ as a development tool:

STEP 1:

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



Step 2:

Select **MCCLPCI_IMP.lib** and add to project

3.2 Using Visual Basic

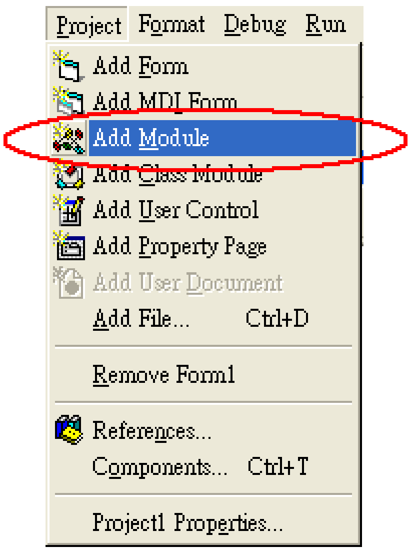
Include Files

MCCLPCI_IMP.bas (for A^+ PC **mode**)

The following is the process of adding the necessary import module **MCCLPCI_IMP.bas** into the project when using an IMP and implementing VB as a development tool:

Step 1

Use **[Add Module]** under **[Project]**



Step 2

Select **MCCLPCI_IMP.bas** and add to the module