IMP Series
Device Driver Library
User Manual

Version: V.1.01

Date: 2013.01

http://www.epcio.com.tw

Table of Content

I. Introduction to the Device Driver Library

The IMP device driver library can be used to drive the intelligent motion control platform (IMP), which was developed using the intelligent motion control chip (IMC).

Based on the functions, the subsequent sections will be categorized as the followings to explain how to use the IMP device driver library.

▲Global Control Interface	Interrupt and reset function settings
▲Pulse Generator Control Interface	Set motion pulse output control
▲Encoder Counter Interface	Set encoder input and counter control
▲Local I/O Control Interface	Local input/output control
▲Remote I/O Interface	Remote input/output control
▲ADC Control Interface	Analog to digital input control
▲PCL Control Interface	Position closed loop control setting
▲DAC Control Interface	Digital to analog output control

Reference Manuals：

Hardware User Manual

IMP-2 Hardware User Manual

IMP-WB-2 Hardware User Manual

Driver User Guide

IMP series Device Driver Library Reference Manual

IMP series Device Driver Library Example Manual

IMP Series Device Driver Library Integrated Testing Environment User Manual

II. Motion Control Card Base Address, Interrupt and Reset Function Setting

The first step in using the IMP Device Driver Library is to initialize the IMP. The following function initialization motion control card can be used：

```
IMC_OpenDevice();
```

If the function return value of IMC_OpenDevice() is TRUE(1), it means that the initialization of the motion control platform is successful, and other functions can be used.

To close the IMP, use IMC_CloseIfOpen(). This function will close all the functional modules in the IMP. If an interrupt is activated when using the IMP , the interrupt vector will also be restored.

The following code shows how to use the IMP Device Driver Library, which uses IMC_GLB_ResetModule() to reset the specified IMC module. This function is usually used in conjunction with the initialization function.

```
if (IMC_OpenDevice())
{
    //    Reset the IMP
    IMC_GLB_ResetModule(RESET_ALL);
    /*
```

```
    The code that the user wants to execute
    */

    IMC_CloseIfOpen();//    Shut down the IMP.
}
```

III. Pulse Output Control

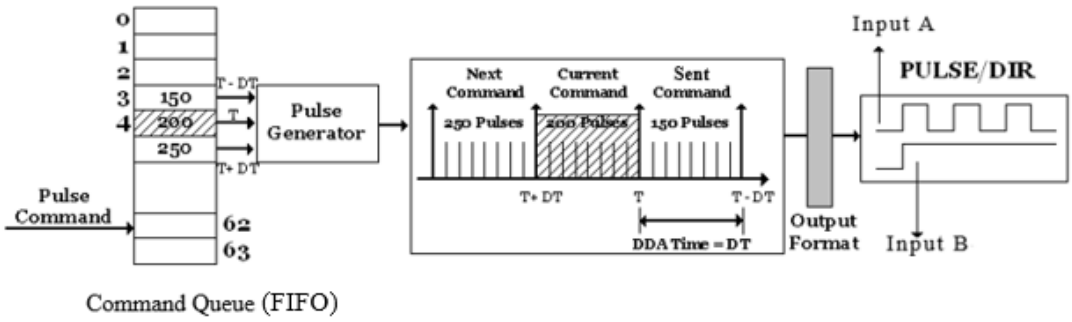
III.1 Basic Pulse Output Control

A set of IMP can equip up to 8 output channels (Channel 0 ~ Channel 7) and use Pulse Generator Engine (PGE) module to control pulse output. This module control mechanism consists of two major steps as follows：

1. Send the pulse command to the pulse command queue (FIFO) of the specified channel. The pulse command queue can store a total of 64 pulse commands.

2. The pulse generator uses IPO time as the time interval (IPO time can be flexibly set). Each time a pulse command from the pulse command queue is automatically read and based on the set output format, these pulses are evenly sent out by the specified channel within the IPO time.

These two steps are shown in the figure below. It is important to note that each output channel has its own command queue. Taking the IMP as an example, it has a total of 8 output channels; hence, there are eight command queues. This figure also shows that each IPO time consumes a pulse command.



As shown in the figure above, the following steps must be completed prior to sending out the pulse command, which include：

1. Use IMC_PGE_SetIPOTime() to set IPO time

2. Set the pulse output format of the specified channel, which includes setting：

   a. Setting pulse output format

      ➔ IMC_PGE_SetOutputFormat()

3. Use IMC_PGE_Start() to start PGE mechanism

   ➔ *See Also* IMC_PGE_Start()

4. Use IMC_PGE_SendPulse() to send pulse command to the command queue of the specified channel

The following code shows how to send a pulse command after initializing the motion control card. In the IMP, when there is position (pulse) or velocity (voltage) command output,

IMC_LIO_SetMotionEnable() is required to enable the output function; some servo systems may need to call IMC_LIO_SetServoOn() to turn on the servo on of the servo motor driver, allowing the system to operate properly. The complete calling procedure is as follows：

```
//    Turn on pulse and velocity motion command output function
IMC_LIO_SetMotionEnable(1);

//    Turn on the servo on variable for Channel 0
IMC_LIO_SetServoOn(0);

//    Set IPO time = 10 ms
IMC_PGE_SetIPOTime(10);

//    Set the output format of Channel 0 to Pulse / Direction format
IMC_PGE_SetOutputFormat(0, PGE_FMT_PD);

//    Start the PGE mechanism
IMC_PGE_Start(1);

//    Send out a pulse command to request Channel 0 to send 200 pulses in one IPO time
IMC_PGE_SendPulse(0, 200);
```

In addition, using IMC_PGE_SetClockDivider() and IMC_PGE_SetClockNumber() can set the upper limit of the total number of pulse that can be sent in each IPO time. By using IMC_PGE_SetIPOTime(), it will automatically calculate and set the upper limit of the total number of pulse with the set IPO time. The total number of pulse included in each pulse command sent by IMC_PGE_SendPulse() must meet this limit.

III.2 Control of Pulse Command Queue (FIFO)

The IMP Device Driver Library provides the following functions to control and read the status of the command queue.

1. IMC_PGE_CheckFifoEmpty() can be used to check whether there is any pulse command stored in the command queue of the specified channel.

2. IMC_PGE_CheckFifoFull() can be used to check whether there is any space available to store pulse command in the command queue of the specified channel. Each command queue has 64 storage spaces.

3. IMC_PGE_GetStockCount() can be used to count the number of pulse commands are stored but not yet executed in the command queue of the specified channel.

The functions as mentioned above can be applied to utilize the space of the command queue fully. Pulse command can be placed into the command queue in advance to avoid the occurrence of motion discontinuity due to insufficient pulse command, improving the operation stability of the system.

III.3 Outputting Pulse Command Control

The IMP Device Driver Library provides the following functions to control and read pulse commands that are being sent out and are no longer in the command queue.

1. IMC_PGE_GetCurrentCommand() can be used to read the pulse commands that are being sent out by the specified channel, including positive and negative signs. From the positive and negative signs of this pulse command, the current direction of motion can be determined.

2. IMC_PGE_SetOutputFormat (specified channel, PGE_FMT_NO) can be used to make the output of the specified channel invalid. When applied, the output of the commands in the command queue and the commands in execution will stop.

III.4 Emergency Stop of Pulse Output

In some cases, the output of pulse must be stopped urgently. The following functions provided by the IMP Device Driver Library can meet this requirement.

1. IMC_PGE_Start(0) can be used to turn off the PGE mechanism, which will stop the output of all channels.

2. If it is required to stop the current command in the command queue, function IMC_PGE_SetOutputFormat can be used to make the output invalid.

III.5 Counting the Number of Output Pulses

The IMP Device Driver Library provides pulse counting functions to obtain the total number of pulses that have been output. These functions include：

1. Use IMC_PGE_EnablePulseCounter() to turn on the pulse count function of the specified channel.

2. Use IMC_PGE_ClearPulseCounter() to zero the value of the pulse counter of the specified channel.

3. Use IMC_PGE_GetPulseCount() to read the value of the pulse counter of the specified channel.

Before using IMC_PGE_GetPulseCount(), it is required to enable the counting function first, which can be done by calling IMC_PGE_EnablePulseCounter(). The following code shows how to read the total number of pulses sent out by Channel 0.

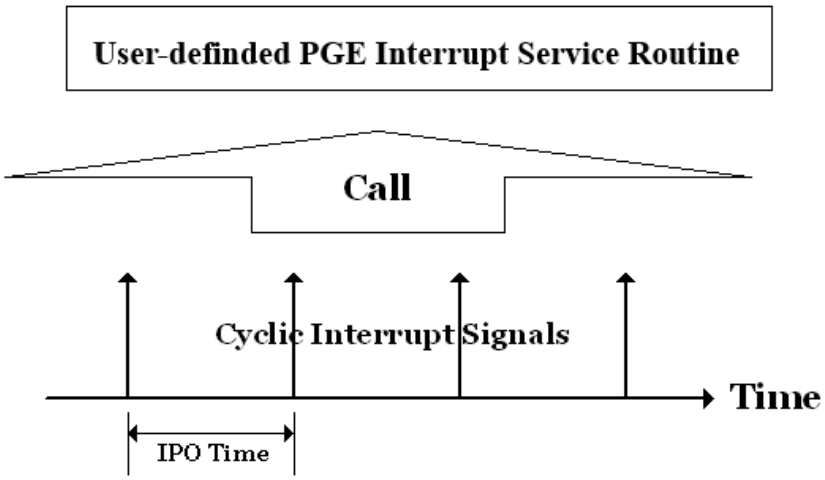
```
//    Clear the value of the pulse counter of Channel 0
IMC_PGE_ClearPulseCounter(0, 1);

//    Enable the pulse counting function of Channel 0
IMC_PGE_EnablePulseCounter(0, 1);
long lPulseCount;
//    Read the total number of pulses sent out by Channel 0
lPulseCount = IMC_PGE_GetPulseCount(0, &lPulseCount);
```

To verify whether the pulse output function is operating properly, the value of the pulse counter is usually compared with the number of pulses actually sent out by using IMC_PGE_SendPulse().

III.6 Cyclic Interrupt Function

The IMP Device Driver Library provides a cyclic interrupt function. When the cyclic interrupt function is enabled, the driver library will use

IPO time as the cycling period. For every IPO time, the user-defined PGE Interrupt Service Routine (ISR) will be triggered automatically as shown in the following figure.



To use the cyclic interrupt function, the following steps must be completed：

1. Declare and define the user-defined PGE ISR. Using the following declaration：

   typedef void(IMC_LIB_CALL *PGEISR)(PGEINT *p);

   Therefore, the user-defined DDA ISR can be defined as follows：

   void ***_stdcall*** PGE_ISR_Function(PGEINT *pstINTSource)

   {

```
        if (pstINTSource->CYCLE)//  Determine whether a cyclic
interrupt has occurred
        {
            /*
            The code to be executed after the cyclic interrupt occurs
            */
        }
}
```

In the PGE ISR, it is necessary to determine whether this function is triggered by a cyclic interrupt. In addition, the declaration of the PGE ISR needs to add the keyword, _stdcall. However, if the standalone mode is used, the keyword, IMC_LIB_CALL, needs to be added.

2. Link PGE ISR using the following function：
   IMC_PGE_SetISRFunction(PGE_ISR_Function)

   After initializing the motion control card, the PGE ISR needs to be linked. The function indicator of the ISR is delivered to IMC_PGE_SetISRFunction().

3. Enable the cyclic interrupt function using IMC_PGE_EnableCycleInterrupt()

➔ *See Also* IMC_PGE_EnableCycleInterrupt()

The following code will show how to use the cyclic interrupt function.

```
void _stdcall PGE_ISR_Function(PGEINT *pstINTSource)
{
        if (pstINTSource->CYCLE)//  Determine whether a cyclic
interrupt has occurred
        {
          /*
          The code to be executed after the cyclic interrupt occurs
          */
        }
       }

if (IMC_OpenDevice())
{
    •
    IMC_PGE_SetISRFunction(PGE_ISR_Function);
    IMC_PGE_EnableCycleInterrupt(1);
    •
}
```

A cyclic interrupt is hardware interrupt with a more accurate

triggering period. It is usually used for tasks that have periodic characteristics and require on-time execution. Using a cyclic interrupt function together with command queue status inspection related functions can ensure that the commands stored in the queue can meet the immediate demand of continuous motion, avoiding the circumstance of motion interruption caused by the lack of command in the queue.

Assuming that a total of 200 pulse commands are required to send out, but the command queue can only store up to 64 commands, the ISR is triggered by using the cyclic interrupt. At this moment, the number of commands currently stored in the queue is read in this function, and the remaining storage spaces are calculated. Such information is then used to determine the commands that can be sent to the queue. These actions will be repeated in the ISR until 200 commands are sent. The code below illustrates this process.

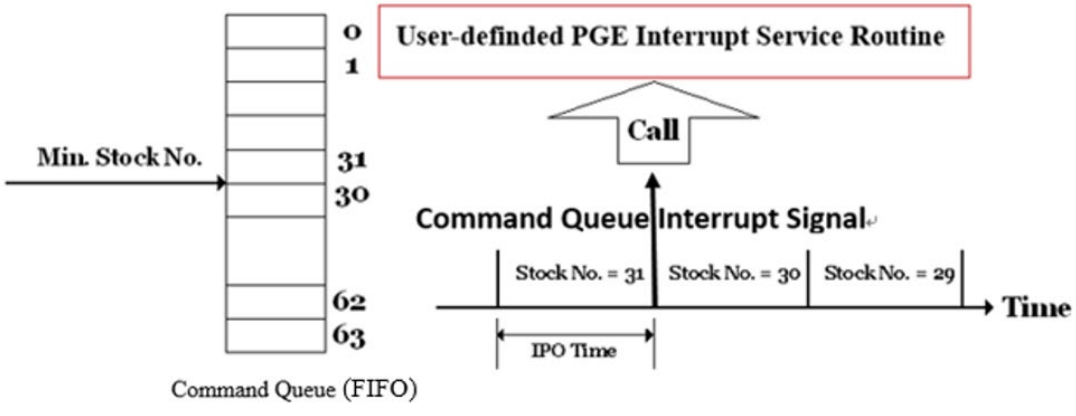
```
int nCount = 200;          //   A total of 200 pulse commands are required
                                to be sent
int nPulse[200] = 150;     //   200 commands, commands can be
                                pre-planned

void _stdcall PGE_ISR_Function(PGEINT *pstINTSource)
{
```

```
        WORD wStockNo;


        if (pstINTSource->CYCLE)//Determine whether cyclic
                                        interrupt has occurred
        {
            if (nCount)// nCount equals 0 when 200 commands are sent
            {
                // Read the number of commands currently stored in
                   the queue of Channel 0
                IMC_PGE_GetStockCount(0, &wStockNo);


                // "i < 64 – wStockNo" because the command queue
                   has only 64 storage spaces
                for (int i = 0;i < 64 - wStockNo && nCount;i++)
                {
                    IMC_PGE_SendPulse(0, nPulse[200 - nCount])
                    nCount--;
                }
            }
        }
}
```

Since cyclic interrupts have the characteristic of periodic occurrence, it is often used to check various states of the system, such as the input and output status of I/O contact point.

III.7 Minimum Stock Interrupt

The IMP Device Driver Library provides command queue minimum stock number interrupt (referred to as minimum stock interrupt) function. When the minimum stock number is set and the minimum stock interrupt function of the specified channel is enabled, the user-defined PGE ISR will be triggered and executed once the command in a queue of the specified channel reaches the minimum stock number as shown in the following figure.



To use the minimum stock interrupt function, the following steps must be completed：

1. Declare and define the user-defined PGE ISR. Using the following declaration：

```
typedef void(IMC_LIB_CALL *PGEISR)(PGEINT *p)
```

Therefore, the user-defined PGE ISR can be defined as follows：

```
void _stdcall PGE_ISR_Function(PGEINT *pstINTSource)
{
    // Determine whether minimum stock interrupt of Channel 0
       occurs
    if (pstINTSource->FIFO0)
    {
        /*
        The code to be executed after the minimum stock interrupt
of Channel 0 occurs.
        */
    }
    // Determine whether minimum stock interrupt of Channel 1
       occurs
    if (pstINTSource->FIFO1)
    {
```

```
        /*
        The code to be executed after the minimum stock interrupt
of Channel 1 occurs.
        */
      }
    .
    .
 }
```

In the PGE ISR, it is necessary to determine whether this function is triggered by the minimum stock interrupt. pstINTSource->FIFO0 ~ pstINTSource->FIFO7 is used to determine whether minimum stock interrupt of Channel 0 ~ Channel 7 occurs. In addition, the declaration of the PGE ISR needs to add the keyword, _stdcall. However, if the standalone mode is used, the keyword, IMC_LIB_CALL, needs to be added.

2. Link PGE ISR using the following function：

   IMC_PGE_SetISRFunction(PGE_ISR_Function);

   After initializing the motion control card, the PGE ISR needs to be linked. The function indicator of the ISR is delivered to IMC_PGE_SetISRFunction().

3. Enable the minimum stock interrupt function of the specified channel using IMC_PGE_EnableStockInterrupt().

   ➔ *See Also* IMC_PGE_EnableStockInterrupt()

The following code shows how to use the minimum stock interrupt function.

```
void _stdcall PGE_ISR_Function(PGEINT *pstINTSource)
{
        // Determine whether command queue interrupt of Channel 0
           occurs
        if (pstINTSource->FIFO0)
        {
            /*
            The code to be executed after the minimum stock interrupt
   occurs
            */
        }
       }
    •
    •
    •
```

```
if (IMC_OpenDevice())
{
    •
   IMC_PGE_SetISRFunction(PGE_ISR_Function);
   IMC_PGE_EnableStockInterrupt(0, 1);
    •
}
```

The minimum stock interrupt is also a hardware interrupt, and the interrupt occurs only when the interrupt condition is met. Compared with the cyclic interrupt that occurs periodically, the minimum stock interrupt would not increase the loading of the system when executing. Usually, the use the minimum stock interrupt function in conjunction with the command queue status inspection related functions can ensure that the commands in the queue will not go under a certain set value (i.e., the command queue minimum stock number), avoiding the circumstance of motion interruption caused by the lack of command in the queue.

Assuming that a total of 200 pulse commands are required to send out, but the queue can only store up to 64 commands, the ISR is triggered by using the minimum stock interrupt. At this moment, the number of commands currently stored in the queue is read in this function, and the remaining storage spaces are calculated. Such information is then used to determine the commands that can be sent to the queue. These actions will be repeated in the ISR until 200 commands are sent.

However, in order to trigger the minimum stock interrupt, it is required to first send out 64 commands to the queue to meet the triggering condition for minimum stock interrupt (that is, when the number of commands stored in the queue is reduced from 31 to 30). The following code illustrates this process.

```
int nCount = 200;          //   A total of 200 pulse commands are required
                                to be sent
int nPulse[200] = 150;     //   200 commands, commands can be pre-
                                planned
 .
 .
for (int i = 0;i < 64;i++) //   Send 64 commands first to the queue of
                                  Channel 0
{
       IMC_PGE_SendPulse(0, nPulse[200 - nCount])
       nCount--;
}

void _stdcall PGE_ISR_Function(PGEINT *pstINTSource)
{
          WORD wStockNo;
```

```
        if (pstINTSource->FIFO0)
        {
            if (nCount)//    nCount equals 0 when 200 commands are
                             sent
            {
                //    Read the number of commands currently stored in
                      the queue of Channel 0
                IMC_PGE_GetStockCount(0, &nStockNo);

                //
                for (int i = 0;i < 64 - nStockNo && nCount;i++)
                {
                    IMC_PGE_SendPulse(0, nPulse[200 - nCount])
                    nCount--;
                }
            }
        }
}
```

IV. Encoder Control

IV.1 Basic Settings and Functions

The IMP has up to 8 channels for inputting encoder signals, namely Channel 0 ~ Channel 7. To use the functions related to encoder control, the following steps must be completed：

1. Use IMC_ENC_SetInputFormat() to set the input signal format of the specified channel. The input signal format must match the hardware setting. When the input signal is the motor encoder feedback signal, please refer to the motor or driver setting. When the regular hand wheel is connected, the input signal format must be the set to A/B Phase (the default setting is A/B Phase input).

2. Use IMC_ENC_SetInputRate() to set the counter signal decoding rate of the specified channel. The decoding rate is valid when the input encoder format is A/B Phase. This function works only if IMC_ENC_SetInputFormat() is set to A/B Phase input.

3. Use IMC_ENC_StartCounter() to enable the counting function of the counter. Before using this function, IMC_ENC_ClearCounter() is usually called first to reset the counter value of the counter to zero.

After completing the above settings, IMC_ENC_ReadCounter() can be used to read the encoder value of the specified channel.

The following code shows how to read the encoder value of Channel 0.

```
long lCounter;
//    Set the input format of Channel 0 to A/B Phase
IMC_ENC_SetInputFormat(0, ENC_TYPE_AB);

//    Set the signal decoding rate of Channel 0 to x4
IMC_ENC_SetInputRate(0, ENC_RATE_X4);

//    Zero the value of the counter of Channel 0
IMC_ENC_ClearCounter(0, 1);

//    Enable the counting function of Channel 0
IMC_ENC_StartCounter(0, 1);

//    Read the value of the counter of Channel 0
IMC_ENC_ReadCounter(0, &Counter);
```

For the needs of wiring, the IMP Device Driver Library provides the following functions to inverse the signal sent to the counter input pin：

1. Use IMC_ENC_EnableInAInverse() to inverse the inA pin of the counter input signal of the specified channel. The default setting is not inversed.

   ➔ *See Also* IMC_ENC_EnableInAInverse()

2. Use IMC_ENC_EnableInBInverse() to inverse the inB pin of the counter input signal of the specified channel. The default setting is not inversed.

   ➔ *See Also* IMC_ENC_EnableInBInverse()

3. Use IMC_ENC_EnableInCInverse() to inverse the inC pin of the counter input signal of the specified channel. The default setting is not inversed.

   ➔ *See Also* IMC_ENC_EnableInCInverse()

4. Use IMC_ENC_EnableInABSwap() to swap the inA and inB pins in the counter input signal of the specified channel before the signal enters the counter. The default setting is no signal swapping required.

   ➔ *See Also* IMC_ENC_EnableInABSwap()

IV.2 Encoder Value Triggered Interrupt Service Function

The encoder value triggered ISR provided by the IMP Device Driver Library (referred to as the encoder value triggered interrupt function) allows the user to set a comparison value for the specified channel. When the function of the specified channel is activated and the encoder value of the channel is equal to the comparison value, the comparator will automatically trigger and execute the user-defined ISR. To use the encoder value to trigger the interrupt function, the following steps must be completed：

1. Define and declare the user-defined ENC ISR, using the following declaration：

   typedef void(IMC_LIB_CALL *ENCISR)(ENC INT *p);

   Therefore, the user-defined ENC ISR can be defined as follows：

```
void _stdcall ENC_ISR_Function (ENCINT *pstINTSource)
{
    // Determine whether the ISR is triggered by the counting
       value of Channel 0
    if (pstINTSource->COMP0)
    {
       /*
```

```
        The code to be executed after the interrupt is triggered by
the encoder value
        */
    }
}
```

In the ENC ISR, it is necessary to determine whether this function is triggered by the encoder value. In the related to encoder ISR, after the encoder value triggers the ENC interrupt function, Channel 0 ~ Channel 7 uses COMP0 ~ COMP7 to determine which channel's encoder value caused the triggering. The relationship between each channel and COMP0 ~ COMP7 is as follows.

Channel 0：pstINTSource ->COMP0 ----

Channel 1：pstINTSource ->COMP1 ----

Channel 2：pstINTSource ->COMP2 ----

Channel 3：pstINTSource ->COMP3 ----

Channel 4：pstINTSource ->COMP4 ----

Channel 5：pstINTSource ->COMP5 ----

Channel 6：pstINTSource ->COMP6 ----

Channel 7：pstINTSource ->COMP7 ----

Therefore, pstINTSource->COMP0 ~ pstINTSource->COMP7 are used to determine whether the ISR is triggered by the encoder

value of the Channel 0 ~ Channel 7. In addition, the declaration of the ENC ISR needs to add the keyword, _stdcall. However, if the Standalone mode is used, the keyword, IMC_LIB_CALL, needs to be added.

2. Link user-defined ENC ISR using the following function：

   IMC_ENC_SetISRFunction(ENC_ISR_Function);

   After initializing the IMP, the ENC ISR needs to be linked. The function indicator of the ISR is delivered to IMC_ENC_SetISRFunction().

3. Set the counter comparison value of the specified channel using IMC_ENC_SetComparator().

4. Use IMC_ENC_EnableComparatorInterrupt() to enable the ISR triggered by the encoder value of the specified channel.

The following code only triggers the ISR after the encoder value in Channel 4 equals the set value.

```
void _stdcall ENC_ISR_Function(ENCINT *pstINTSource)
    {
        // Determine whether it is triggered by the encoder counting
```

```
            value of the 4th channel
        if (pstINTSource->COMP4)
        {
            /*
            The code to be executed after the encoder value triggers the
    ISR
            */
        }
    }
 .
 .
if (IMC_OpenDevice())
{
     .
    IMC_ENC_SetISRFunction(ENC_ISR_Function);
    //    Set the counter comparison value of Channel 4 to 10000.
    IMC_ENC_SetComparator(4, 10000);

    // Enable the ISR triggered by the encoder value of Channel 4.
    IMC_ENC_EnableComparatorInterrupt(4, 1);
     .
}
```

IV.3 Index Interrupt

The IMP Device Driver Library provides the encoder index interrupt function that triggers a user-defined ISR when the encoder's index (Z Phase) signal is input. To use the index interrupt function, the following settings must be completed：

1. Define and declare the user-defined ENC ISR. Using the following declaration：

   typedef void(IMC_LIB_CALL *ENCISR)(ENC INT *p);

   Therefore, the user-defined ENC ISR can be defined as follows：

```
void _stdcall ENC_ISR_Function (ENCINT *pstINTSource)
{

    // Determine whether it is triggered by the index signal of the
       channel 0
    if (pstINTSource->INDEX0)
    {
        /*
        The code to be executed after the index interrupt occurs
        */
```

```
        }
}
```

In the ENC ISR, it is necessary to determine whether this function is triggered by the index interrupt. In the encoder-related interrupt function, after the index interrupt occurs, Channel 0 ~ Channel 7 uses INDEX0 ~ INDEX7 to determine which channel's index signal caused the triggering. The relationship between each channel and INDEX0 ~ INDEX7 is as follows.

Channel 0：pstINTSource ->INDEX0 ----

Channel 1：pstINTSource ->INDEX1 ----

Channel 2：pstINTSource ->INDEX2 ----

Channel 3：pstINTSource ->INDEX3 ----

Channel 4：pstINTSource ->INDEX4 ----

Channel 5：pstINTSource ->INDEX5 ----

Channel 6：pstINTSource ->INDEX6 ----

Channel 7：pstINTSource ->INDEX7 ----

Therefore, pstINTSource->INDEX0 ~ pstINTSource->INDEX7 are used to determine whether the index interrupt of Channel0 ~ Channel7 occurs.

In addition, the declaration of the ENC ISR needs to add the keyword, _stdcall. However, if the Standalone mode is used, the keyword,

IMC_LIB_CALL, needs to be added.

2. Link ENC ISR using the following function：

   IMC_ENC_SetISRFunction(ENC_ISR_Function);

   After initializing the motion control card, the ENC ISR needs to be linked. The function indicator of the ISR is delivered to IMC_ENC_SetISRFunction().

3. Enable the index interrupt triggering function of the specified channel using IMC_ENC_EnableIndexInterrupt().

   ➔ *See Also* IMC_ENC_GetIndexStatus()

The following code only enables the index interrupt function of Channel 5.

```
void _stdcall ENC_ISR_Function(ENCINT *pstINTSource)
{
    // Determine whether the function is triggered by the index interrupt
       of the Channel 5
    if (pstINTSource->INDEX5)
    {
          /*
          The code to be executed after the index interrupt occurs
```

```
            */
        }
    }
    .
    .
if (IMC_OpenDevice())
{
        .
        .
    IMC_ENC_SetISRFunction(ENC_ISR_Function);
    //    Enable the index interrupt triggering function of Channel 5
    IMC_ENC_EnableIndexInterrupt(5, 1);
        .
        .
}
```

IV.4 Counter Value Latch Function

The IMP Device Driver Library provides the counter value latch function. Users can set the triggering signal source. These triggering signal sources are used to trigger the action of recording the value of the counter in the latch register. Users can then use the function provided by the driver library to read the value recorded in the latch register.

The triggering signal source of the counter triggering latch function

is divided into two categories, namely the index signal triggering or the external condition triggering. To use the counter latch function, users need to apply first IMC_ENC_Set IndexLatchSource () or IMC_ENC_SetExternalLatchSource () to set whether index signal or external condition is the triggering signal source. The function is declared as follows：

```
void IMC_ENC_SetIndexLatchSource(    WORD Channel,
                                     WORD Source    );
void IMC_ENC_SetExternalLatchSource ( WORD Channel,
                                      WORD Source );
```

Channel indicates the numbering of the channels, ranging from 0 to 7.Source indicates the triggering source. There are eight types of triggering sources to trigger the latch counter value separately for index signal triggering and external condition triggering. When setting, multiple conditions can be combined. These triggering signal sources include：

NO_TRIG_ENC	No triggering signal source selected
INDEX0_TRIG_ENC	Index signal of Channel 0 encoder
INDEX1_TRIG_ENC	Index signal of Channel 1 encoder
INDEX2_TRIG_ENC	Index signal of Channel 2 encoder
INDEX3_TRIG_ENC	Index signal of Channel 3 encoder

INDEX4_TRIG_ENC	Index signal of Channel 4 encoder
INDEX5_TRIG_ENC	Index signal of Channel 5 encoder
INDEX6_TRIG_ENC	Index signal of Channel 6 encoder
INDEX7_TRIG_ENC	Index signal of Channel 7 encoder
OTP0_TRIG_ENC	Channel 0 positive limit switch input signal
OTP1_TRIG_ENC	Channel 1 positive limit switch input signal
OTP2_TRIG_ENC	Channel 2 positive limit switch input signal
OTP3_TRIG_ENC	Channel 3 positive limit switch input signal
OTP4_TRIG_ENC	Channel 4 positive limit switch input signal
OTP5_TRIG_ENC	Channel 5 positive limit switch input signal
OTP6_TRIG_ENC	Channel 6 positive limit switch input signal
OTP7_TRIG_ENC	Channel 7 positive limit switch input signal
OTN0_TRIG_ENC	Channel 0 negative limit switch input signal
OTN1_TRIG_ENC	Channel 1 negative limit switch input signal
OTN2_TRIG_ENC	Channel 2 negative limit switch input signal
OTN3_TRIG_ENC	Channel 3 negative limit switch input signal
OTN4_TRIG_ENC	Channel 4 negative limit switch input signal
OTN5_TRIG_ENC	Channel 5 negative limit switch input signal
OTN6_TRIG_ENC	Channel 6 negative limit switch input signal
OTN7_TRIG_ENC	Channel 7 negative limit switch input signal

When the triggering signal source setting is completed, the value of the counter will be recorded in the latch register when these signals occur. However, before starting the counter latch function using

IMC_ENC_StartCounter(), users need to call IMC_ENC_SetCounterLatchMode() to set the latch mode. This function is declared as follows：

```
IMC_ENC_SetCounterLatchMode(    WORD Channel,
                                WORD Mode)
```

Channel indicates the numbering of the channel, ranging from 0 to 7. Mode is the latch triggering mode, which includes：

ENC_TRIG_FIRST	When the triggering condition is met for the first time, the value of the counter is latched and will no longer change.
ENC_TRIG_LAST	When the triggering condition is met, the value of the counter is latched. However, a new value will be latched if the new condition is met again.

Users can use IMC_ENC_ReadLatchCounter() to read the value recorded in the latch register of the specified channel.

The following code shows how to set the triggering source to the index signal of the encoder connected to Channel 0, and read the latched value when the index signal occurs. Users can obtain the real position of the index signal from the latched value.

```
void _stdcall ENC_ISR_Function(ENCINT *pstINTSource)
{
    // Determine whether the trigger is made by the index signal of the
       0th channel
    if (pstINTSource->INDEX0)
    {
        // The code to be executed after the index interrupt occurs
          long lLatchValue

        // Read the value in the latch register
        IMC_ENC_ReadLatchCounter(0, &lLatchValue)
    }
}
.
.
if (IMC_OpenDevice())
{
      .
      .
     IMC_ENC_SetISRFunction(ENC_ISR_Function);
     // Set the index signal of Channel 0 encoder as the triggering source
        for the latch counter of Channel 0.
     IMC_ENC_SetIndexLatchSource(0, INDEX0_TRIG_ENC);
```

```
    // Set the latch triggering mode of the Channel 0 latch counter to
       continuous triggering
    IMC_ENC_SetCounterLatchMode(0, ENC_TRIG_LAST);

    .
    .
}
```

V. Local Input and Output (Local I/O) Control

There are 42 local input and output which can be used as the input and output (I/O). These I/O have been designed on the IMP for specific applications, including：

The inputs have a total of 25 ports, which include：

Home Sensor	8 inputs
Limit Switch Positive(+)	8 inputs
Limit Switch Negative(-)	8 inputs
Status for Emergency Stop	1 input

The outputs have a total of 17 ports and 1 special output for motion enabled, which include ：

Servo On/Off	8 outputs
LED Light	8 outputs
Enabling Position Ready	1 output
Motion Enable	1 output

The following section will explain how to use these local inputs and outputs.

V.1 Basic Settings and Functions

The local digital input and output are sorted into groups with eight ports in each group. The 40 inputs and outputs are divided into OT+, OT-, HOME, SERVO and LED according to the pre-defined applications. The OT+ / OT- / HOME are the inputs in default, while the SERVO / LED are the outputs in default. The default conditions of all the inputs and outputs are set to disable (0, no output and input).

Taking OT+ as an example, IMC_LIO_GetPlusLimitLDIInput() can be used to read the digital signal input value of OT+ 0 ~ OT+ 7. This function is declared as follows：

```
IMC_LIO_GetPlusLimitLDIInput(DWORD *LDIState);
```

Bit 0~Bit 7 of *LDIState obtained by this function represents the input status of OT+ 0 ~ OT+ 7, while Bit 8 ~ Bit 31 has no meaning.

Therefore, if the input value obtained by using IMC_LIO_GetPlusLimitLDIInput(&dwInput) is 0x0002, it means that the current digit input value of OT+1 is 1, since 0x0002 in terms of the binary system is equivalent to 0b0000000000000010. The value of the bit position associated with OT+1 is 1.

➔ *See Also* IMC_LIO_GetMinusLimitLDIInput()
IMC_LIO_GetHomeSensorLDIInput()

When it is required to use the input and output of the functions as mentioned above, the meaning of each port can be found in the following table.：

IMP local input and output (Local I/O)

LIO	Definition	Corresponding SCSI II position	Note
0	Channel 0 OT+	10（100Pin External input）	Able to trigger an interrupt
1	Channel 1 OT+	60（100Pin External input）	Able to trigger an interrupt
2	Channel 2 OT+	14（100Pin External input）	Able to trigger an interrupt
3	Channel 3 OT+	64（100Pin External input）	Able to trigger an interrupt
4	Channel 4 OT+	18（100Pin External input）	Able to trigger an interrupt
5	Channel 5 OT+	68（100Pin External input）	Able to trigger an interrupt

6	Channel 6 OT+	7（40Pin External input）	Able to trigger an interrupt
7	Channel 7 OT+	8（40Pin External input）	Able to trigger an interrupt
8	Channel 0 OT-	11（100Pin External input）	Able to trigger an interrupt
9	Channel 1 OT-	61（100Pin External input）	Able to trigger interrupt
10	Channel 2 OT-	15（100Pin External input）	Able to trigger an interrupt
11	Channel 3 OT-	65（100Pin External input）	Able to trigger an interrupt
12	Channel 4 OT-	19（100Pin External input）	Able to trigger an interrupt
13	Channel 5 OT-	69（100Pin External input）	Able to trigger an interrupt
14	Channel 6 OT-	9（40Pin External input）	Able to trigger an interrupt
15	Channel 7 OT-	10（40Pin External input）	Able to trigger an interrupt
16	Channel 0 HOME	9（100Pin External input）	Able to trigger an interrupt
17	Channel 1 HOME	59（100Pin External input）	Able to trigger an interrupt
18	Channel 2 HOME	13（100Pin External input）	Able to trigger an interrupt
19	Channel 3 HOME	63（100Pin External input）	Able to trigger an interrupt
20	Channel 4 HOME	17（100Pin External input）	Able to trigger an interrupt
21	Channel 5 HOME	67（100Pin External input）	Able to trigger

			an interrupt
22	Channel 6 HOME	5（40Pin External input）	Able to trigger an interrupt
23	Channel 7 HOME	6（40Pin External input）	Able to trigger an interrupt
24	Channel 0 SERVO	12（100Pin External output）	
25	Channel 1 SERVO	62（100Pin External output）	
26	Channel 2 SERVO	16（100Pin External output）	
27	Channel 3 SERVO	66（100Pin External output）	
28	Channel 4 SERVO	20（100Pin External output）	
29	Channel 5 SERVO	70（100Pin External output）	
30	Channel 6 SERVO	11（40Pin External output）	
31	Channel 7 SERVO	12（40Pin External output）	
32	Channel 0 LED	Non-external signal output	Conditional triggering
33	Channel 1 LED	Non-external signal output	Conditional triggering
34	Channel 2 LED	Non-external signal output	Conditional triggering
35	Channel 3 LED	Non-external signal output	Conditional triggering
36	Channel 4 LED	Non-external signal output	Conditional triggering

37	Channel 5 LED	Non-external signal output	Conditional triggering
38	Channel 6 LED	Non-external signal output	Conditional triggering
39	Channel 7 LED	Non-external signal output	Conditional triggering
40	ESTOP	57（100Pin External input）	
41	P_RDY	58（100Pin External output）	
42	MOTION ENABLE	Non-external signal output	

The IMP has already designed the specific use of these inputs and outputs. If the IMP-WB-1 or IMP-WB-2 is used, the following functions can be used to perform reading or output operations for the corresponding inputs and outputs on the IMP-WB-1 or IMP-WB-2.

The IMP Device Driver Library provides the following functions to read the status of the inputs.

1. Use IMC_LIO_GetHomeSensorStatus() to read the HOME status of the specified channel. When the HOME status changes, no interrupt signal is generated. Only this function can be used to check the HOME status of the specified channel.

2. Use IMC_LIO_GetPlusLimitStatus() to check whether the specified channel has touched the positive limit switch of the hardware. If it is, the equipment may be in danger of collision, and the user should take emergency responses immediately. When the positive limit switch of the hardware is touched, the user-defined ISR will be triggered.

3. Use IMC_LIO_GetMinusLimitStatus() to check whether the specified channel has touched the negative limit switch of the hardware. If it is, the equipment may be in danger of collision, and the user should take emergency responses immediately. When the negative limit switch of the hardware is touched, the user-defined ISR will be triggered.

The IMP Device Driver Library provides the following functions to set the status of the output.

1. Use IMC_LIO_SetServoOn() to enable the servo drive of the specified channel. This channel is connected to the servo drive port of the motor drive. When this function is called, the specified channel can receive the position or velocity commands from the IMP.

2. Use IMC_LIO_SetServoOff() to turn off the servo drive of the specified channel. This channel is connected to the servo drive port of the motor driver. When this function is called, the specified

channel will no longer receive the position or velocity commands. After the initialization function is successfully called, the default status is to turn off the servo drive function.

3. Use IMC_LIO_SetLedLightOn() to set the LED output of the specified channel. The LED output is connected to the LED indicator on the IMP. When this function is called, the specified channel is able to accept LED output commands from the IMP. After the initialization function is successfully called, the default LED status is off.

4. Use IMC_LIO_SetMotionEnable() to enable the position (pulse) and velocity (voltage) command output function of the IMP. When this function is called, the output function will be enabled. After the initialization function is successfully called, the default status of the output function is off.

The output of the IMP has specific applications; however, these outputs can also be used for general output purposes. For example, some channels use stepping motors and do not require servo on/off signal control. In this case, the servo on/off outputs of these channels can be used for general output purposes.

V.2 Hardware Limit Switch Interrupt

The IMP 's positive limit switch and negative limit switch (or over-travel limit switch) and Home Sensor (home position or origin position) provide hardware limit switch interrupt function (referred to as the limit interrupt). When the situation of reaching a limit switch occurs, the user-defined ISR will be triggered, and users can use this function to plan the emergency responding actions.

To use the limit interrupt function, the following steps must be completed：

1. To define and declare the user-defined LIO ISR, the declaration of the LIO ISR must follow the definitions stated below：

   typedef void(IMC_LIB_CALL *LIOISR)(LIOINT *p);

   Therefore, the user-defined LIO ISR can be defined as follows：

   ```
   void _stdcall LIO_ISR_Function(LIOINT *pstINTSource)
   {
       //   Determine whether a limit interrupt occurs
       if (pstINTSource-> OTP0)
       {
          /*
          Emergency responding action when the over-travel limit
   ```

```
occurs
            */
        }
    }
```

In the LIO ISR, use OTP 0 ~ OTP 7 / OTN 0 ~ OTN 7 / HOME 0 ~ HOME 7 to determine whether this function is triggered by the limit interrupt. The definition of OTP 0 ~ OTP 7 / OTN 0 ~ OTN 7 / HOME 0 ~ HOME 7 are described as follows：

a. IMP positive limit switch (OT+)

pstINTSource-> OTP0 Channel 0’s OT+

pstINTSource-> OTP1 Channel 1’s OT+

pstINTSource-> OTP2 Channel 2’s OT+

pstINTSource-> OTP3 Channel 3’s OT+

pstINTSource-> OTP4 Channel 4’s OT+

pstINTSource-> OTP5 Channel 5’s OT+

pstINTSource-> OTP6 Channel 6’s OT+

pstINTSource-> OTP7 Channel 7’s OT+

b. IMP negative limit switch (OT-)

pstINTSource-> OTN0 Channel 0’s OT-

pstINTSource-> OTN1 Channel 1’s OT-

pstINTSource-> OTN2 Channel 2’s OT-

pstINTSource-> OTN3 Channel 3's OT-

pstINTSource-> OTN4 Channel 4's OT-

pstINTSource-> OTN5 Channel 5's OT-

pstINTSource-> OTN6 Channel 6's OT-

pstINTSource-> OTN7 Channel 7's OT-

c. IMP home sensor

pstINTSource-> HOME0 Channel 0's home sensor

pstINTSource-> HOME1 Channel 1's home sensor

pstINTSource-> HOME2 Channel 2's home sensor

pstINTSource-> HOME3 Channel 3's home sensor

pstINTSource-> HOME4 Channel 4's home sensor

pstINTSource-> HOME5 Channel 5's home sensor

pstINTSource-> HOME6 Channel 6's home sensor

pstINTSource-> HOME7 Channel 7's Home sensor

In addition, the declaration of the LIO ISR needs to add the keyword, _stdcall. However, if the standalone mode is used, the keyword, IMC_LIB_CALL, needs to be added.

2. Link user-defined LIO ISR

After initializing the IMP, the LIO ISR needs to be linked. The function indicator of the ISR is delivered to IMC_LIO_SetISRFunction(), as follows：

IMC_LIO_SetISRFunction(LIO_ISR_Function);

3. Use IMC_LIO_SetPlusLimitTriggerMode() / IMC_LIO_SetMinus LimitTriggerMode()/IMC_LIO_SetHomeSensorTriggerMode() to set OTP 0 ~ OTP 7 / OTN 0 ~ OTN 7 / HOME 0 ~ HOME 7 interrupt triggering mode; namely the upper limit triggering, lower limit triggering or transition triggering.

4. Use the IMC_LIO_EnablePlusLimitInterrupt() / IMC_LIO_EnableMinusLimitInterrupt()/IMC_LIO_EnableHomeSens orInterrupt() to enable the limit interrupt function.

The following code shows how to use the limit interrupt.

```
void _stdcall LIO_ISR_Function(LIOINT *pstINTSource)
{
    //    Determine whether a limit interrupt occurs
    if (pstINTSource-> OTP0)
    {
        /*
        Emergency responding action when the over-travel limit occurs
        */
```

```
    }
}
.
.
if (IMC_OpenDevice())
{
      .
      .
    IMC_LIO_SetISRFunction(LIO_ISR_Function);
     //    Enable OTP0 interrupt triggering function
    IMC_LIO_SetPlusLimitTriggerMode(LIO_OTP0, LIO_INT_FALL);
    IMC_LIO_EnablePlusLimitInterrupt(LIO_OTP0, 1);
      .
      .
}
```

V.3 Timer Timing Interrupt

The IMP provides a 32-bit timer. Users can set the timer, and when the set time is up, timer interrupt of the timer (referred to as the timer interrupt) will be triggered, and the timer will restart. This process will continue until the user switches off this function. To use the timer interrupt, the following steps must be completed：

1. To define and declare the user-defined timer ISR, the timer ISR must follow the definitions stated below：

```
typedef void(IMC_LIB_CALL *TMRISR)(TMRINT *p);
```

Therefore, the user-defined timer ISR can be defined as follows：

```
void _stdcall Timer_ISR_Function(TMRINT *pstINTSource)
{
    //    Determine whether a timer interrupt occurs
    if (pstINTSource->TIMER)
    {
        /*
        The code to be executed when the set time has ended
        */
    }
}
```

In the timer ISR, pstINTSource->TIMER must be used to determine whether this function is triggered by a timer interrupt. In addition, the declaration of the timer ISR needs to add the keyword, _stdcall. However, if the standalone mode is used, the keyword, IMC_LIB_CALL, needs to be added.

2. To link user-defined timer ISR

   After initializing the motion control card, the timer ISR needs to be linked. The function indicator of the ISR is delivered to IMC_TMR_SetISRFunction(), as follows：

   IMC_TMR_SetISRFunction(Timer_ISR_Function);

3. Use IMC_TMR_SetTimer() to set the timer. The unit for the timer is based on the system clock (10ns).

4. Use the IMC_TMR_SetTimerIntEnable() to enable the timer interrupt function.

   ➔ *See Also* IMC_TMR_GetTimerIntEnable()

5. Use the IMC_TMR_SetTimerEnable() to enable the timer timing function.

   ➔ *See Also* IMC_TMR_GetTimerEnable()

The following code shows how to use the timer interrupt function.

```
void _stdcall Timer_ISR_Function(TMRINT *pstINTSource)
{
    //    Determine whether a timing interrupt occurs
```

```
    if (pstINTSource->TIMER)
    {
        /*
        The code to be executed when the set time has ended
        */
    }
}
.
.
if (IMC_OpenDevice())
    {
     .
     .
    IMC_TMR_SetISRFunction(Timer_ISR_Function);
    //    Set the time of TMR timer to 10ns x 1000000 = 10ms
    IMC_TMR_SetTimer(1000000, 0);
    IMC_TMR_SetTimerIntEnable(1);    //    Enable the timer interrupt
    function
    IMC_TMR_SetTimerEnable(1);       //    Enable the timer
     .
     .
    }
```