PDL程式開發手冊

第1.0版
2011年12月06日

HIWIN Mikrosystem Corp.

此頁空白

目錄

修訂記錄

版次	日期	適用範圍	註記
1.0	2011.12.06	D系列驅動器	初版發行

1: 前言

PDL(Process Description Language)是一套可用來開發mega-fabs驅動器內運動控制程式的語言。使用者可透過PC來撰寫*.pdl檔(文字檔)的PDL程式，並透過MDP(DSP 韌體)來執行暫存記憶體中的PDL程式。
PDL的主要特性如下：

- 多工(Multitasking) – 最多可同時處理4個任務(task)。
- 使用者可在程式中定義自己的變數。
- 提供陣列和指標的功能。
- 可透過Loop、while、if、else、till 和goto等命令來建立控制迴路。
- 可建立數學運算式。
- 程式的記憶體大小可達到128Kbytes。
- 使用者所定義的變數記憶體大小可達到16000 long words。
- 使用者變數(user variable)名稱之長度最多17個字元。
- 標籤(label)名稱的長度最多24個字元。
- 程序(proc)名稱的長度最多24個字元。

標題 **頁碼**

1.1: 多工

PDL提供同時執行4個task的功能。在DSP的記憶體中，每個task都擁有屬於自己的計數器、堆疊指標、旗標和堆疊空間。假設所有的task的優先權都一樣時，當有n個task要執行，且不在sleep或stop的狀態時，每個task將在n個phase(66.7μsec)中執行一次。下面將介紹如何改變task的優先權，使用者有權限決定重置(reset)之後task0到task3的執行順序，可藉由在程式中寫入**#task/n**方式來達成此功能，但並非所有的task都需要在重置之後才執行。

全域變數對於所有的task是共用的，相同的命令也可在不同的task中執行，因此若要保護全域變數、全域程式碼或者其他資源被兩個以上的task存取時，可利用Lock 、Unlock命令來管控資料的存取。另外多工的同步化(multitask synchronization)也可透過Lock 、Unlock命令來達成。

Task可透過下列方式來執行：

1. 將**#task/n**指令放置在程式的任一個段落時，當程式執行到那個段落時，編號n的task將被執行，並可透過ret的命令來終止編號n的task。
2. 主機端可利用run命令來建立一個新的task(可參照mpi.dll檔案中的RunFuncPdlN和SetArrayRunFuncN。在使用run命令時必須包含task的標籤位址，標簽名稱必須在前頭加上 _ 的符號，並利用halt命令來終止。同一個標籤在主機端只能執行一次。

Task可透過下列三種方式來終止：

1. 執行 **halt/ret/rertprintl**命令，其中halt命令只能在舊版的PDL中執行。
2. 可在task中利用kill命令來關閉其它task。
3. 可由主機端(PC)來終止，參照mpi.dll檔案中的killTask。

1.1.1: 定義Task的優先權

task_prior[n]=k可以用來定義task的優先權。其中n介於0到3代表task的編號，k為task的執行時間，單位為phase，若k小於等於0，則k視為1。因此可以定義所有task的執行時間如下：

T = total execute cycle = Σ task_prior[n].

其中n為編號0到3中排除sleep或stop狀態，並且執行中的task，T的單位為phase。

編號n的task執行速率為:

rate = samp_rate * task_prior[n] /T

如果沒有設定task的優先權，則task彼此之間的優先權式相同的。

下列例子將解釋如何改變task的優先權：

```
_my_critical_task:                    // 標籤的起始
……
task_prior[task_num]=5;      // 變更目前task的優先權
…….
…..
task_prior[task_num]=0;      // 目前task的優先權設回預設值
….
```

Notes:

當task處於sleep或stop的狀態時，將不考慮其優先權。

當task處於lock的狀態時，除非執行解除lock的狀態，否則將不考慮其他task的優先權。

1.2: 變數的型別

變數的屬性：

1. 可定義為系統變數或使用者變數
2. 型別(Type):**long，float** (32 bit)，**short** (16 bit)， **state** (1 bit)， **Float pointer以及long pointer** (32 bit)。
3. 容量(Size):所有的變數可以宣告為陣列形式，其最小的陣列元素數目為1。
4. 視野(Scope):宣告在程序(procedure)標頭與程序主體內的使用者變數，視為暫存在此程序的區域變數。

系統變數:

為預先定義的變數(名稱、大小 和位置)，每個系統變數都代表特定的功能。例如X_vel_max、X_acc、X_dcc 和X_new_sm_fac等系統變數，定義了運動速度軌跡，當指定X_trg新的數值給驅動器時驅動器將依照設定的速度軌跡移動到X_trg所設定的位置。

使用者變數:

以#long、#short和#float的指令宣告在PDL的檔案中。使用者變數也可被宣告為long 或float型別的指標。使用者變數可視為暫存的區域變數，其記憶體容量可為16位元(short) 或32位元(long/float/pointer)。

狀態(State);

為系統變數且被使用在條件敘述中，其佔用的記憶體容量為1bit。可透過seton、setoff、toggle和state assignment等命令來修改狀態。狀態可代表數位輸入、輸出訊號或內部狀態。所有的狀態變數屬於狀態陣列(status array)變數中的某個位元(bit)。

一個型態為short或long的變數在不同的使用場合可以用來表示為有帶符號或無帶符號的數值。容量(size)的屬性是有關於陣列型態。陣列內的元素可使用[]來存取，例如:

```
vel_max[2]=20000;
abspos[n]=50000;          // n是一個short型態的區域變數
```

變數可被常數或short/long型態的變數所索引，且不管變數的容量(size)為何，它都可以被索引。

Example:

```
#short Eli, Rafi, Benny ,
My_array[100], N;          // 宣告一個short型態的使用者變數與陣列
Eli[0]=15;                 // 指定 Eli=15
Eli[1]=100;                // 指定Rafi=100
Eli[2]=500;                // 指定Benny=500
N=55;
My_array[N]=Benny;         // 指定 My_array[55]=500
```

1.2.1: 指標

指標為32bit的變數，用來儲存變數的位址，例如：

```
#long v, *pv;      // pv 為一個指標，指向一個long或short型態的變數位址
pv=&v;             // pv 指向變數v的位址
*pv=55;            // 指定v=55
```

其中， * 為間接值運算子(indirect value operator)， 只有在變數宣告為指標時才能使用 。意思為指標變數的值為某個位址上所儲存的數值。運算子 &為取址運算子，用來取得某個變數的位址。指標變數可宣告為long 或 float型別的變數。例如:

```
#float   f, *pf;     // pf 為一個指標，指向float型態的變數
pf= &f;
*pf=5.4;             //指定 f=5.4
```

指標不可被宣告為short型態。 存取short型態的變數可用宣告為long型態的指標。指標也可被下列幾種方式索引：

```
#long v[6],n, *pv;
pv=&v;
*pv=0;      // v[0]=0
*pv[1]=10; //v[1]=10;
pv=pv+2;
```

```
*pv=20;     // v[2]=20
pv=&v[3];
*pv=30;     // v[3]=30
n=4;
pv=&v[n];
*pv=40;     // v[4]=40;
n=1;
*pv[n]=50;   // v[5]=50, note that operator * perform before operator []
*pv[1]=50;   // v[5]=50;
```

使用指標之前必須指向任一變數的位址當作初始化，且指標的型別必須與存取資料的型別相同，否則會有不能讀寫的情形發生。指標亦可為程序(procedure)的回傳值。

1.3: 程序

程序類似C or C++裡面的函式(function)，與C語言不同的是程序沒有回傳值，但可利用指標變數來達成類似回傳值的功能。

語法:

proc <程序名稱>(<變數型態> 變數名稱, <變數型態> 變數名稱,.....) do

.....

程序主體

end; // 程序結尾

Example:

// 下列的程序用來計算三個數值的和並由sum指標變數傳回結果。

```
proc add3( long a, long b, long c, long * sum) do
        #long tmp;    // 宣告一long型態的區域變數
        tmp=a+b;
        *sum=tmp+c;
end;                        // 返回程序的呼叫位址
…….
#long v1,v2, s; // 宣告型態為long的全域變數
add3(4,5,6,&s);    //  結果儲存於s=15;
v1=10;
v2=100'
add3(v1,v2,1,&s);  //結果儲存於s=111;
```

註:

- 程序透過關鍵字**proc**來宣告。
- 程序的主體可撰寫在**do**和**end**之間。
- 程序中宣告的變數為暫存變數，如同C/C++中的區域變數，因此程序中宣告的變數，並不能在程序外面使用，不過宣告在程序外的非暫存變數類似全域變數，卻可以在程序中使用。
- 程序只可判讀自己區域內定義的標籤(Lables)，並不能判讀全域標籤或其他程序中的標籤
- 由於每個task都有屬於自己的堆疊空間，因此相同程序可同時被多個task執行，但全域變數不在此範疇內。
- 暫存變數的型別可為long、float 或者可為指標變數，但不能為short型別。
- 呼叫程序的語法與C/C++相似，首先定義程序名稱再來在 ( )中宣告變數。
- 當呼叫程序時，編譯器會先判斷程序中的變數型別和個數是否與宣告的程序一樣。程序不能像C++的函式一樣設定預設值或者多載程序。
- 程序中的暫存變數名稱可與使用者定義的全域變數同名或其他程序中的暫存變數同名。
- 在使用程序時，可以宣告程序的原型來避免程式編譯的錯誤發生。例如：

```
proc mul( float a, long b, float * res);   // 宣告程序原型
        #float f1, m; // 宣告為全域的使用者變數
        f1=44.2;
        mul(f1, 20, &m);    //  程序返回後的結果 m=884;

proc mul( float a, long b, float * res) do   // 程序的定義
        *res=a*b;
end;                        // 返回至程序呼叫位址
        …….
```

2: 命令

命令可分成4個主要的部份：

1. **指定(Assignment)：**包含數學運算和函式.
2. **流程控制(Program Flow)：**Goto, call, run, ret, exitproc,if, else, while, till, loop, wait and halt.
3. **前置處理：** 與編譯有關的命令。
4. **特殊命令：printl/prints/reprint**

Note:

- 當宣告命令時必須以 '**;**'做結尾。
- **標籤(Lable)**必須以'**:**'做結尾。
- 註解必須在' // 浮號之後 ，或者 包含在 /*.... */裡面。

2.1: 指定

指定可為下列形式：

1. var = var/常數;
2. var = {const_0 , const_1 , const_2, ... , const_n};
3. var =”string…”;
4. var = -var;
5. var = <state_name> ; var = <label_name> ;
6. var = var op var/constant;
7. var = func_name(arg_1,arg_2,...arg_n);
8. state = 條件敘述式

變數的宣告型別可為short，long和float，使用者可依照需求作各種變數型別的宣告。
將long型別的資料指定給short型別的變數將會有部分資料遺失。若將short型別的資料指定給long型別的變數，資料會以帶符號型態(singed)的增加。
若float變數的內容超出long或short的變數時,將具有float變數的內容指定給long或short的變數將是不正確的。
常數可視為字元碼。

運算子:

+	兩運算元相加
-	兩運算元相減
*	兩運算元相乘
/	兩運算元相除
&	邏輯運算子AND(交集)，只有兩個運算元的比較結果是true時，才回傳true，其餘情況皆傳回false。
|	邏輯運算子OR(聯集)，只有兩個運算元的比較結果都是false時，才傳回false，其餘情況回傳true。
@	邏輯運算子XOR(互斥或)，兩個運算元的比較結果都是true或false時，就傳回false；兩個運算元的比較 結果一個是true而另一個是false時，就傳回true。
%	取得餘數

複合命令運算子：

+=	兩運算元相加後再指定給左變數
-=	兩運算元相減後再指定給左變數
*=	兩運算元相乘後再指定給左變數
/=	兩運算元相除後得到商數再指定給左變數
&=	兩運算元做位元and運算後再指定給左變數
|=	兩運算元做位元or運算後再指定給左變數
@=	兩運算元做位元xor運算後再指定給左變數
%=	兩運算元相除得到餘數後再指定給左變數

可利用複合命令運算子，來縮短程式的撰寫，並使程式執行的更有效率，例如var1=var1+var2 可寫成var1+=var2。

Example :

```
#short x,y,ar[100],YS,DN,EL;          // 宣告short型態的使用者變數/陣列
#long   p1,p2,pr[60];        // 宣告long型態的使用者變數/陣列
#float f1,f2;

f1=0,5;
f2=f1*4000;
p1=600000;
p2=p1;                       // 將p1的值指定給p2
pr[5]=p1;                    // 將p1的值指定的pr陣列內的第5個元素
pr=400;                      // 等同於 pr[0]=400;
pr=0x02FC;                   // 將常數用十六進制指定給pr[0]
x=7; y=5;
pr[x]=pr[y]+400000;          // 將px陣列的第7個元素的內容加上400000後的結果指定給pr陣列
                             //  的第5個元素
x=x & y;                     // 將7與5做AND後，指定給x
ar[x]=ar[2] * ar[y];
ar[x]=ar[x] * ar[y];
wait 20;
p2=p1+100000;
ar={1,3,5,6,900,-8000};              //初始化陣列內容。請參考本頁尾陣列初始化的註記。
ar[2]={1,3,5,6,900,-8000};           // 從陣列的第2個元素開始初始化。請參考本頁尾陣列初始化的註
記。
YS={100,200,300};                    // 等同於YS=100; DN=200; EL=300;
x+=y;   // x=x+y
x*=0.5; //x=x*0.5
```

註：

陣列初始化可從陣列的某個元素為起始並指定一連串的值，依序指定的特定之陣列元素內容。陣列的初始化一次的最多只能使用7個數值指定給陣列。

例如:

```
#long array[100];
array [ 0 ] = {10, 11, 12, 13, 14, 15, 16};    // 從array陣列的第0個元素開始至第6個元素，
                                                // 依序指定數值。array[0]=10, array[1]=11, …
                                                //  array[6]=16。
array [ 7 ] = { 17, 18, 19, 20, 21, 22, 23 };
```

2.1.1: 自動遞增的索引值

陣列中的索引可自動遞增，例如：

```
#short n,p;
#long myar1[10],myar2[10];
…….
n=0; p=0;
loop (10) do
myar1[n+]=myar2[p+];   // 將myar2陣列的第p個變數指定給 myar1陣列內的第n個變數後，p與
                       // n索引值遞增1。
end;
```

註:

陣列在取得值之後，索引值會自動往上遞增

當來源陣列的索引值遞增之後，目標陣列的索引值才會跟著遞增，因此如果想利用一個索引值(n)來宣告目標和來源陣列時，則必須以下列的例子為指定陳述式：

```
myar1[n+]=myar2[n];   //將myar2陣列的第n變數指定給 myar1陣列內的第n個變數後，索引值n
                      // 遞增1。
```

2.1.2: 自動遞增減

可利用++或- -運算子來遞增或遞減 long 或 short 型別的變數：

```
#long *po, myvar[10];          //宣告 long 型態的指標和陣列
myvar ++;                      // myvar = myvar +1;
myvar --;                      //myvar = myvar -1;
myvar [4]++;
myvar [n]++;
po=&myvar;
*po++;        // 等同於myvar++;
*po[1]++;     // 等同於myvar[1]++;
*po[n]++;     // 等同於myvar[n]++;
```

2.1.3: 指定字串

字串可以指定給陣列：

```
ar = "this is a "
```

在這個例子中每4個字元將儲存在ar變數的陣列中，因此當需要儲存的字串很長時，建議使用數個陣列來儲存字串。下面的例子定義一個long型別的陣列，並透過lcdout的變數去做輸出的動作；字串的長度限制為12個byte。

```
#long str[50], n,len;
str = "This is a lo";
str[6] = "ng string";
len=6;              // 字串的大小為24個字母(6 long words)
call out_lcd;
……
                    // 此副程序輸出str所指到的字串，其大小為24個字母。
out_lcd:
```

```
n=0;
  loop(len) do
     lcdout = str[n];                  // 以LSB byte的型式輸出
     lcdout = str[n]/256;              // 以 2個byte的型式輸出
     lcdout = str[n]/65536;            // 以3個byte的型式輸出
     lcdout = str[n]/16777216;         // 以MSB byte的型式輸出
     n= n+1;
  end;
ret;
```

2.1.4: 指定狀態位址

狀態的位址亦可以指定給變數，只要將狀態名稱寫在<>中，並指定給變數即可。

Example:

a1=<x_run> ; // a1的變數將儲存x_run的狀態
狀態位址<x_run>可視為常數。
a1變數也可被命令 **seton/setoff/toggle**使用，或可使用於條件敘述。
Example:

```
#long a1;
a1=<y1_run>;
till ( <a1>);
a1= <anysensor>;
seton <anysensor>;
```

2.1.4.1指定狀態給變數

可直接使用sttovar命令，將狀態或條件敘述直接指定給變數。

範例:

```
#long var1[5],var2;
#float f1;
var1= sttovar s2; // 指定狀態
var1[var2]= sttovar ~s2;
var1[3]= sttovar (s2 & ~s3) | (~s2 &s3);
f1 = sttovar s1 | ( s3 & v2>1000 ) ;
var1 = s1 &   var2->3   & var2->10
```

如果邏輯條件成立，則此變數被設為1；反之，則設為0。

2.1.5: 指定標籤位址

將標籤(Label)位址指定給變數時，可將標籤名稱寫在< >中，再將標籤位址指定給變數。

Example:

```
a1=<_y1_init> ; // a1的值將被指定為標籤_y1_init的位址
```

如上所示，<**_y1_init** >視同一個常數，且將標籤位址指定給a1變數。

2.1.6: sizeof

運算子sizeof可用來回傳陣列大小，例如：

```
#long myarray[275], onevar;
```

var = sizeof(myarray); // 將陣列的大小指定給變數 var (var=275)
var = sizeof(onevar); // 將陣列的大小指定給變數 var (var=1)
註：
此語法與C/C++相似；但是sizeof在這裡會回傳陣列的大小，在C語言中則是回傳物件的大小(byte數)。

2.2: 程式流成命令(Program Flow Commands)

2.2.1: Halt

格式: halt;

當task執行到halt的命令時，它將停止執行。此時task是處於閒置的狀態，並且可執行其他外部或內部命令。

2.2.2: wait/Sleep

格式: wait <區域變數或常數 >或Sleep <區域變數或常數 >

等待命令用來在某一段時間內暫停task的執行。延遲時間的計算方式如下:
Sleep命令: value * 0.001sec
wait命令: value * 0.00005 sec
wait和Sleep命令差別在於使用Sleep命令，並不會使得該task佔用DSP處理時間，因此其它的task執行起來會更有效率。

Examples:
#short dd,x,ar[20];
Sleep 1000; // 等待1000×0.001=1秒
dd=2000;
Sleep dd; // 等待2000×0.001=2秒
ar[4]=5000; ar[5]=7000;
x=5;
wait ar[4]; // 等待5000×0.000005=0.25秒
Sleep ar[x]; // 等待7000×0.001=7秒

註: 應避免在lock/unlock命令間使用Sleep命令，若真的需要暫停task時，可使用wati命令。
另外，Sleep/wait 命令並不能直接使用指標變數，例如：

```
#long *ps,t
ps=&anyvar;
Sleep *ps;
```

可用下列方式取代:

```
t=*ps;
sleep t;
```

2.2.3: 前往(Goto)

格式: goto <標籤名稱> ;

Goto命令將目前程式的執行轉換到標籤所定義的特定程式區域。
該標籤可用25字元內的任何名字的字串做命名，其結尾必須以冒號 **(:)**做結尾。

Example:
....
goto loca1;// 將目前的執行轉移到標籤所定義的程式區
....
loca1: // 標籤所定義的程式區之起始位置

2.2.4: 間接前往(Indirect Goto)

格式: goto <標籤名稱>[<變數/常數>];
Example:
#short indx;
.....
goto label_tab[2]; *// 程式的執行跳到func2*
.....
indx=3;
goto label_tab[indx]; *//程式的執行跳到func2*
label_tab:
goto func0;
goto func1;
goto func2;
goto func3;

2.2.5: 呼叫與返回(Call and Ret)

格式: call <標籤名稱>;

Call命令可將程式的執行序轉換到指定的標籤中，並且將目前執行的位置放到堆疊裡面。Ret命令可返回至呼叫此副程序時的位置(從堆疊出取回該位置)。

Example:
call *subr1;*
…
subr1:
…
ret*;*
Call命令所使用的個數上限，取決於task的堆疊大小。其中每使用一次call命令，則減少1個堆疊空間，呼叫程序(procedure)，則會佔用更多堆疊空間，取決於程序中變數的多寡。

註 :
當任務(task)透過外部程式來執行時(例如:mpi)，則可透過ret命令來終止任務。若任務(task)透過pdl執行#task/n時，則只能利用halt命令來終止任務。 如果要將參數載入副程序中，則應使用procedure的方式來達成。

2.2.6: 離開程序(exitproc)

格式:exitproc;

通常程式執行到end時，則表示程序結束。如果要在程序中跳離時則必須使用exitproc的命令。

Example:

```
proc func1( long flag) do
    ……
    if(flag=-1) exitproc;        // 在程式中跳離程序func1
    ……
    end;                         //程序終點
```

2.2.7: 迴圈(Loop)

格式: loop(<區域變數或常數>) do

…

end;

Loop命令將執行do到end之間的程式n次，其中n以short或常數的區域變數宣告。如果n=0時，loop命令將執行65536次。

Example:

```
nk=20;
loop(nk) do          // 執行區間中的程式20次
    ….
    pos1=pos2;
    loop(100) do     // 將執行區間中的程式100次
        ….
        ….
     end;
end;
```

loop命令使用的個數取決於每個task中可使用的堆疊空間。

2.2.8: IF and Else

格式:

1. **if (條件敘述) 命令;**
2. **if (條件敘述) do**

```
        ….
    end;
```

3. **if (條件敘述) do**

```
        ….
    else do
        ….
    end;
```

條件敘述是由一個或數個邏輯判斷所組成，如下所示：

(Condition1 [&/| contidion2] [&/| contidion3]....)

條件敘述可利用( )的運算子來概括起來，並且可利用~的運算子來反態。另外，格式1中並不允許wait，state，if，while，till和loop等命令被使用。

條件敘述可為下列型式：:

1. <state>
2. ~<state>
3. < <var> >
4. < ~<var> >
5. ~(條件敘述)

在例子3和4中的變數用來儲存狀態的位址。

條件敘述的關係可以下列方式去表示：

= (相等)
> (大於)
< (小於)
>= (大於等於)
<= (小於等於)
<> (不等於)
TOUT (時間判斷運算子，2.2.11節有詳細的說明)

If-else所能使用的迴圈層數目被限制在200個，其中可包含loop和while命令。另外運算子**&**為邏輯的AND，運算子|為邏輯的OR，運算子~的邏輯反相。

Example:

```
 pos2=-100000;
if(pos1>100000 | pos1<pos2) do
        if (pos1>1000000) do
                abspos=pos2;
     else do
                abspos=-pos2;
         end;
        if(LEFT_LIM)   call sub1;
 end;
#short st;;
st= <X_LL>;
if (   <st> ) pos2=pos2+100;      // 如果左極限信號觸發，則執行pos2=pos2+100
if(~<st> ) pos2=pos2-100;         // 反之，則執行pos2=pos2-100
if (pos>1000) call func1;         // 如果條件成立，則呼叫func1
if (pos>1000) goto label1;        // 如果條件成立，則執行label1
if (pos>1000) func1(pos,300);     // 如果條件成立，則呼叫func1(pos,300)
if (pos>1000) sleep 100 ;         // 如果條件成立，則待時0.1sec
```

註:

- Else命令對於if命令等同於end命令，可用來終止if命令之do的區塊，並且層級與if同階層。
- 格式1的寫法不需要寫入else命令。
- 格式1中，如果條件敘述(condition expression)和命令(command)的長度低於緩衝區的長度，則if命令將在50usec中便可執行完畢。

2.2.9: While

格式: While (條件敘述) do

```
    ....
end;
```

條件敘述可依據if-else命令中所提到的格式去撰寫，當條件式成立時，則while命令將執行do到end區塊內的程式。迴圈層使用的上限為200個，可包含loop 和 if-else等命令。

2.2.10: Till

格式: till (條件敘述);

till 命令將暫停task的執行直到條件成立時，才繼續往下執行。

Example:

```
till(x_ref_pos[0]>20000 & ~x_run);    // 直到x_ref_pos大於20000時，並且馬達停止時，
                                      // 才執行之後的命令。
```

2.2.11: TOUT condition

格式: if(…….. v1 TOUT v2/常數)
till(…….. v1 TOUT v2/常數)
while(…….. v1 TOUT v2/常數)

變數v1和v2的宣告必須為整數型式。除此之外TOUT的條件與其它條件運算子(=, >, <, <=, >=, <>)有相同的語法。如果以下條件成立時，則為TRUE：

fclk-v1 >=v2

其中，fclk 是系統的時間變數，以15000Hz的頻率去記數。使用者可利用~運算子將條件敘述反態，例如：**till(~(v1 TOUT v2));** //等到條件敘述fclk-v1<v2成立。
以下例子可以了解此運算子的使用情形：

```
#long timeout,tend;
timeout=60000;
x_jvl=2000;                  // 以JOG模式使馬達運動
tend=fclk+timeout;           // 計算執行JOG模式的終止時間
till ( x_home | fclk>=tend ); //直到歸原點成功或fclk大於3 sec後才執行後續命令。
x_stop_m=1;                  // 停止馬達運動
if (fclk>=tend ) do
      printl/101("ERROR time out when search home limit");
else do
      printl/101("OK, found home limit");
end;
```

fclk的數值是從2^{31}-1到-2^{31}的區間來計數。因此若依照上面till命令中的條件敘述可能會有發生執行錯誤的風險產生，為了解決此問題可用if命令來取代，因此程式可修改為：

```
#long timeout,t0, tmp;
timeout=60000;
x_jvl=2000;       // 以JOG模式使馬達運動
t0=fclk;          // 紀錄起始時間
waitloop
tmp=fclk-t0;    // 計算從t0開始經過了多少時間
if ( ~x_home & tmp < timeout ) goto waitloop; //如果歸原點失敗且尚未timeout，則移至迴圈執行
x_stop_m=1;  // 停止馬達運動
if (tmp>=timeout) do
     printl/101("ERROR time out when search home limit");
else do
     printl/101("OK, found home limit");
end;
```

此程式即使fclk設定在2^{31}-1到-2^{31}之間執行上也不會有任何問題，不過卻需要多寫goto waitloop這行命令使程式延續下去，為了修正此問題，可將程式再修改成下列型式：

```
#long timeout, t0;
timeout=60000;
x_jvl=2000;
t0=fclk;  // 紀錄起始時間
till ( x_home | t0 TOUT timeout );  //持續等待至歸原點成功或時間超過3秒
x_stop_m=1;
if ( t0 TOUT timeout ) do
```

```
    printl/101("ERROR time out when search home limit");
else do
    printl/101("OK, found home limit");
end;
```

2.3: 內建函式(Build in Function)

函式為內建的處理程序，將輸入的引數進行處理，並將結果儲存於變數之中。函式也用來表示為指定陳述式的另一種型態，一般的用法如下：

<var>=func_name(arg_1,arg_2,...arg_n);

其中，arg_1... arg_n必須為區域變數，且其最後一個引數也可以是常數。

2.3.1: Max and Min

格式: <var>=max(<var>,<var/constant>);
<var>=min(<var>,<var/constant>);

MAX和MIN函式可取代"if , else"陳述式，其方式如下所示：

if(var1>var2) do var3=var1; else do var3=var2; end;

可由下列程式片段來取代：

var3=max(var1,var2);

另，下列的程式片段:

if(var1>var2) do var3=var2; else do var3=var1; end;

可由下列程式片段來取代：

var3=min(var1,var2);

2.3.2: Abs

格式: <var>=abs(<變數/常數>);

這個函式為取得變數的絕對值，表示方式如下：

if (var2>0) do var1=var2; else do var1=-var2; end;

判斷式可取代為下列形式：

var1=abs(var2);

2.3.3: Unsign

格式:<var1>=unsign(<var2/constant>);

此函式可將var2變數不帶符號的變數，再指定給var1，也可將short型別變數轉換成不帶符號的short型態的變數，再指定給long型別變數。如下列所示：

```
#long lng1;
#short srt1;
srt1= -1;
lng1= srt1;                 // 將-1指派給lng1變數
lng1= unsign(srt1);         // 將65535指派給lng1變數
```

2.3.4: Sign

格式:<var1>=sign(<var2/constant>);

如果引數為正值則結果是1，如果引數為負值則結果是-1，如果該引數為零則結果是0。

2.3.5: sin,cos

格式:<var1>=sin(<var2/constant>);
<var1>=cos(<var2/constant>);

將引數的單位為徑度。

2.3.6: sqrt

格式:<var1>=sqrt(<var2/constant>);

將傳入的引數開根號後指定給左變數。若引數為負值的，var1的數值也是為負的。

2.3.7: divi

格式: <var1>=divi(<var2>, <var3/constant>);

var1可得到var2/var3的整數部分

以下例子將比較divi函式與標準除法運算子的差別。

```
#long r, p;
#float f;
p=11;
f=p/4;   // f=2.75
r=p/4;   // r=3 ( 四捨五入後得到的結果)
f=divi(p,4); // f=2
r=divi(p,4); // r=2
```

2.3.8: shift

格式:<var1>=shift(<var2>, <var3/constant>);

此函式將var2位移n個bits，若n>0則往左位移，n<0則往右位移。

Example:

```
#long r1,r2,rs1, rs2;
r1=0x000000F0;   //11110000
rs1=1; rs2=-2
r2=shift(r1, rs1);   // r2將被設成111100000
r2=shift(r1, rs2);   // r2將被設成111100
r2=shift(r1, -8);    // r2將被設成0
r2=shift(r1, -4);    // r2將被設成1111
r2=shift(r1, 0);     // r2將被設成11110000
r2=shift(r1, 4);     // r2將被設成111100000000
r2=shift(r1, 8);     // r2將被設成1111000000000000
r2=shift(r1,12);     // r2將被設成0x000f0000(Hex)
r2=shift(r1, 24);    // r2將被設成0xf0000000(Hex)
r2=shift(r1, 27);    // r2將被設成0x80000000(Hex)
```

2.3.9: bitset,bitclr,bittog

格式:<var1>=bitset(<var2>, <var3/constant>);
<var1>=bitclr(<var2>, <var3/constant>);
<var1>=bittog(<var2>, <var3/constant>);

bitset函式可設定var2中第n個bit值為1。
bitclr 函式可設定var2中第n個bit值為0。

bittog函式可切換var2中第n個bit值為1或0。
Example:
#Long v1,v2
v1= bitset (v1, 0); // v1=0x00000001
v1= bitset (v1, 1); // v1=0x00000011
v1= bitclr (v1, 0); // v1=0x00000010
v2=31;
v1= bitset (v1, v2); // v1=0x80000002(Hex)
v1= bittog (v1, 4); // v1=0x80000012(Hex)
v1= bittog (v1, 4); // v1=0x80000002(Hex)
v1= bittog (v1, v2); // v1=0x00000010

2.3.10: memcpy

格式:<var1>=memcpy(<var2>, <var3/constant>);
var1 – 目標變數/陣列
var2 – 來源變數/陣列
var3 – n =要複製之變數的數目(必須為long/short型別)

此函式將var2陣列複製給var1陣列。透過此函式來複製陣列會比利用迴圈的方式來指派陣列要快上許多。此函式並不會進行資料型別的轉換，因此var1和var2的資料型別必須皆為float、long或者short。

Example (將陣列IN1複製給IN2):
#long IN1[3],IN2[3];
IN1={0x01020304,0x11223344,0xaabbccdd}; // init array IN1
IN2=memcpy(IN1,3); // 將IN1的陣列複製給IN2
halt;

註:

var3的大小並沒有特別限制，但需注意不可大於varr1和var2的陣列的大小。如果var3的值小於等於0時，此函式並不會有任何效用。

2.3.11: memset

格式:<var1>=memset(<var2>, <var3/constant>);
var1 – 目標變數/陣列
var2 – 來源變數值
var3 – n =設定的變數的數目(必須為long/short型別)

此函式可將陣列var1的內容設成var2的值。透過此函式來設定var1陣列的數值，會比利用迴圈的方式來指派陣列要快上許多。此函式並不會進行資料型別的轉換，因此var1和var2的資料型別必須皆為float、long或short。
Example (清除陣列IN1):
#long IN1[3],t1;
t1=0;

IN1=memset(t1 ,3); //清除陣列IN1

halt;

註：

var3的大小並沒有特別限制，但需注意不可大於varr1和var2的陣列的大小。如果var3的數值小於等於0時，此函式並不會有任何效用。

2.3.12: memmin,memmax

格式:<var1>=memmin(<var2>, <var3/constant>);

<var1>=memmax(<var2>, <var3/constant>);

var1 – 目標變數
var2 – 來源陣列
var3 – n =判斷陣列個數的數目(必須為long/short型別)

memmin/memmax函式可用來判斷陣列中的最大/最小值。透過memmin/memmax函式來判斷陣列的最大最小值，會比利用迴圈的方式執行此功能要快上許多。此函式會自動做資料型別的轉換，因此var1和var2的資料型別不一定要相同。

Example
#long l1[3],mina,maxa;
l1={4,11,8};

mina=memmin(l1 ,3); // mina=4

maxa=memmax(l1 ,3); // maxa=11

halt;

註：

var3的大小並沒有特別限制，但需注意不可大於varr1和var2的陣列的大小。如果var3的數值小於等於0時，此函式並不會有任何效用。

2.3.13: memsum

格式:<var1>=memsum(<var2>, <var3/constant>);

var1 – 目標變數
var2 – 來源陣列
var3 – n =被相加的陣列數目(必須為long/short型別)

此函式可求得陣列相加的總合。透過此函式來計算陣列的總合，會比利用迴圈的方式來求得要快上許多。此函式會自動做資料型別的轉換，因此var1和var2的資料型別不一定要相同。

Example (找平均值):
#long l1[3], avr;
l1={2,11,8};
avr=memsum(l1 ,3); // avr=21
avr=avr/3; // avr=7
halt;

註：

var3的大小並沒有特別限制，但需注意不可大於varr1和var2的陣列的大小。如果var3的數值小於等於0，此函式並不會有任何效用。

2.4: 特殊命令(Special Commands)

2.4.1: printl/retprintl

printl/retprintl命令的格式：

printl / mode1 /mode2 (“ …. String…..”,var1,…varN);
retprintl / mode1 /mode2 (“ …. String…..”,var1,…varN); // printl and ret

mode1 – 8字元的16進位參數，可用來定義顏色、蜂鳴聲和樣式等輸出型式。
mode2 – 8字元的16進位參數，可用來定義事件資訊。

printl/reprintl等命令’( )’中的格式與C語言相同。Retprintl命令結合了ret和printl兩種命令的功能。此命令可確保當task被終止時，訊息也可同時傳送給主機端(host)。

註：

prints/printl/retprintl等命令可顯示的最多變數為8個，最少為0個。

Examples:
printl / 00000103 / 00000001 (“ This is a test”);
printl / 00000103 / 00000002(“ position of axis 2 is: %g ”,ref_pos[2]);
printl / 00000103 / 00000003(“ position of axis 2 is: %g ”,ref_pos[2]);

2.4.2: printl/retprintl + parameters

PDL version 2+後的版本支援改良版的printl命令，可將參數以即時的方式送出去。
printl命令支援:

任何變數/狀態可被顯示出來，其中包含指標或陣列定義的變數、狀態和區域變數。

Examples
```
#long *pn , n, ar[10];
pnn=&pnn;
n=2;
…..
printl/103(“*pnn=%08x , n=%ld , n address is %08x , ar[2]=%ld ”,*pnn, n, &n, ar[n]);
printl/103(“state IN1=%d ”, IN1);
proc func(long v1, float f2 , instate run) do
printl/103(“local variables: v1=%ld, f1=%g , state=%ld ”,v1,f1, <run>);
end;
ar="this is a test";  //將14個字元指派給陣列
printl/103(" the string is: %s ",ar[0],ar[1],ar[2],ar[3]); // 顯示字串
```

註:

當使用者要正確的將字串顯示出來時，必須提供相對應的陣列大小給字串，其中一個long型別的陣列可以儲存4個字元。

使用者可以利用%g來顯示浮點變數、%x來顯示整數型的16進位變數、%d和%u分別用來顯示singed/unsigned的十進位變數、%s可用來顯示字串。Print命令敘述格式與C語言和C++語言中的printf相同。

Important: 當使用者要連續撰寫兩個以上的識別碼(%x、%d、%u、%s)時，必須利用空白鍵隔開，以避免顯示錯誤的訊息。

1. 參數是以即時的方式傳遞。
2. printl/retprintl命令的語法在新版的PDL中沒有做任何改變，須注意的是使用者只能使用%g來顯示浮點型式的變數，並不能顯示double型式的變數。

2.4.2.1Example

此範例由取得的旗標值去切換printl顯示的訊息。

```
proc printMsg(long flag, long p1, float p2) do
        if(flag<0 | flag>=5 )do
             printl/101("flag=%ld out of orange 0...4",flag);
             exitproc;
        end;
flag=flag*5;              // printl命令+2個變數+exitproc命令的程式碼長度為5。
goto prlabel[flag];   //依據旗標(flag)值去執行prlabel中所相對應的動作。
prlabel:
            printl/103("MSG A, p1=%ld p2=%g ",p1,p2); exitproc;
            printl/103("MSG B, p1=%ld p2=%g ",p1,p2); exitproc;
            printl/103("MSG C, p1=%ld p2=%g ",p1,p2); exitproc;
            printl/103("MSG D, p1=%ld p2=%g ",p1,p2); exitproc;
            printl/103("MSG E, p1=%ld p2=%g ",p1,p2); exitproc;
end;
```

在此範例中flag的程式碼長度=printl+間接參數+exitproc 的程式碼程度 = 5

Examples:

```
_a1:
printMsg(0,17,4.2);
ret;
_a2:
printMsg(1,ltest1,ff1);
ret;
_a3:
printMsg(ltest2,ltest1,ff1);
ret;
```

2.4.3: Set,Clear,Toggle,State

語法: **seton <state_name | <var> >**

setoff <state_name | <var> >

toggle <state_name | <var> >

seton/setoff/toggle等命令可用來修改狀態變數的狀態，如果狀態變數是輸入/內部的狀態，則狀態被修改的時間相當短暫。這些命令主要用在修改輸出狀態變數的狀態。

Example

```
seton vacum_sl   // 打開 vcaum_sl狀態
sleep 1000
setoff vacum_sl   // 關閉vacum_sl狀態
```

2.5: 指令(Directive)

指令是在編譯時執行，所有的指令前面都會加上' **#** '。

2.5.1: 任務(Task)

格式: **#task/<n>;**

當此命令用特定的編號來初始task。n代表task的編號。
Example:
p1=p2;

halt;

#task/2; // 在驅動器重新啟動之後，task2將在此執行。

call initt2;

當驅動器重新啟動之後task將被指令(#, directive)來執行，並可透過halt/ret命令來終止。

2.5.2: Long/Short/Float

格式: **#short** [_ro] [_lp(v1,v2)] **str_1,str_2,....str_n;**
#long [_ro] [_lp(v1,v2)] **str_1,atr_2,....str_n;**
#float [_ro] [_lp(v1,v2)] **str_1,atr_2,....str_n;**

其中，str_1...str_n可宣告為任何形式的字串(上限為15字元，另外不能將字串宣告為驅動的內部變數)。當變數設定為陣列時可在[]中設定陣列的大小。

Example：
#long u,k,yosi,my_array[200],tmp_array[800];
#float _ro v1; // v1唯讀
#float _lp(-0.5, 0.5) pgain; // pgain的範圍值介於-0.5~0.5之間

_ro和_lp變數通常使用在唯讀或範圍設定的情形。此種變數設定只用在KMI存取變數的情形中。這些指令可用來宣告使用者變數。對於每個使用者變數編譯器都會在驅動器中定義一個位址來暫存變數資料。#long、#float和#short在程式中可以被宣告很多次，當編譯成功之後，浮點變數會以32bit的資料長度存放在USER_n.VRS檔案中。編譯器會將浮點變數存放在29到1的位址中。當使用者要宣告指標變數時，必須在變數前面加上'*'。

Example:

#long v1, *pvv;
#float ff, *pff, *pfp;
沒有short型別的指標，使用long型別來宣告指標時，也可以指向short型別的位址。

2.5.3: Define

格式: **#define str_a str_b**
#define str_a(arg1,arg2…argn) longstring

在第一種格式中，編譯器會將str_a取代為str_b。此種用法適合用來定義常數。
第二種格式可以使用在定義函式類型的巨集(此種語法適用在PDL25+版本以上的編譯器)。Longstring的定義型式可用str_a中宣告的arg1,arg2…argn來定義。巨集的命名也可用已經定義的巨集來定義。

Example1：
#define ADD2(r,a,b) r = a+b;
#define ADD3(r,a,b,c) ADD2(r,a,b) r += c;
#define ADD4(r,a,b,c,d) ADD3(r,a,b,c) r += d;
#long result,var1,var2,var3,var4;
…..
ADD4(result,var1,var2,var3,var4) // result=var1+var2+var3+var4

此外，巨集可使用數行敘述來定義，但必須在每一行敘述的最末端加上'\'，如下列範例:

Example2：

```
#define LONGCODE   var1=var2+var3;\
                   ff1=ff2*ff3;\
                   someproc(var1,varb)\
                        call somelabel;\
                   var++;
....
LONGCODE    //執行巨集LONGCODE
```

註：

如果在巨集的最後結尾處加上';'，則會將';'視為str_b的一部分，如下面範例所示:

```
Example
#define size 10;
#long array[size]
```

陣列大小的定義會以[10;]的型式取代，因此會造成編譯的錯誤，所以要避免在巨集命名的最後端加上';'。 正確的使用方法如下所示:

```
#define size 10
```

#define 、#undef、#ifdef、#else和#endif等命令的編譯優先順序，跟C、C++編譯器所定義的順序相同。

2.5.4: Undef

格式:**#undef string**

移除#define所定義的巨集。

```
Example
#define KEY   STRING1
……
#undef KEY
#define KEY   STRING2
```

2.5.5: Ifdef ifndef else elifdef elifndef endif

格式:#ifdef KEY

```
        code a
    #endif
```

如果定義了KEY則編譯code a，否則不執行。

```
#ifdef KEY
    code a
#else
    code b
#endif
```

如果定義了KEY則編譯code a，否則編譯code b。

```
#ifndef KEY
    code a
#else
    code b
#endif
```

如果沒有定義KEY，則編譯code a，否則編譯code b。

使用者可使用#elifdef或#elifdef來接續#ifdef/ifndef。

Example：
```
#ifdef KEY1
    code a
#elifdef KEY2
    code b
#endif
```

如果KEY1已經定義，則code a將被編譯，或如果KEY2已被定義，則code b將被編譯。

當宣告了#ifdef/ifndef，都必須宣告#endif來結尾。另外可宣告巢狀的#ifdef。其中code a和code b也可使用其他指令(directives)/巨集(macros)來宣告，例如#define、#undef、#ifdef等命令。

2.5.6: Include

格式: **#include <filename>**

當編譯器執行到#include <filename>這命令時，編譯器將開始編譯指定的檔案。當指定的檔案編譯完畢之後，編譯器將回歸到原始的編譯檔案中。此命令最多可將檔案放在20個層級的資料夾中。例如:

#include "..\common.h"